

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

本书详解了Android音视频开发相关技术，从原理到案例展示了音视频开发的独特魅力，希望帮助读者在Android音视频开发的道路上不断进步。

Broadview[®]
www.broadview.com.cn

Android 音视频开发

何俊林 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>





何俊林，曾就职于爱奇艺公司，先后参与过TV播放器业务和需求开发工作，以及TV新播放内核的开发和维护工作。主要研究方向为多媒体、音视频、Codec相关方向。长期在CSDN上坚持写博客，2016年获得CSDN音视频之星、年度博客之星等称号。热爱开源，乐于研究和分享技术。同时运营公众号“何俊林”，该公众号有4万多人关注。



微信公众号



Android 音视频开发

何俊林 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING





内 容 简 介

近年来,直播、短视频行业的相关业务发展迅猛,本书主要介绍其中涉及的 Android 音视频开发相关技术。本书一共有 11 章,分别介绍了音视频基础知识、MediaPlayer、MediaPlayerService、StagefrightPlayer、NuPlayer、OpenMAX 框架、FFmpeg 项目、FFmpeg 源码分析及实战、直播技术、H.264 编码及 H.265 编码、视频格式分析内容。希望本书能帮助读者系统学习、化繁为简,在 Android 音视频开发的道路上不断进步。

本书适合具有一定 Android 开发基础并且对音视频技术方向感兴趣的读者阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Android 音视频开发 / 何俊林著. —北京: 电子工业出版社, 2018.11
ISBN 978-7-121-34996-6

I. ①A… II. ①何… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字(2018)第 206231 号

责任编辑: 付 睿

印 刷: 三河市良远印务有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 29.25 字数: 655 千字

版 次: 2018 年 11 月第 1 版

印 次: 2018 年 11 月第 1 次印刷

定 价: 99.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:(010) 51260888-819, faq@phei.com.cn。



前言

从来没想到自己能出一本书。

写书是一件很考验人耐心的事情，从打算写一本书开始，我心里每时每刻都像有一块大石头压着一样。一要保证专业性，二要保证质量，同时还要考虑怎么表达才能让别人明白自己的意思，所以写书并没有那么简单。

近年来，直播、短视频行业的相关业务发展迅猛，很多人希望学习其中涉及的 Android 音视频开发相关知识，而 Android 音视频开发的难度相对较高，这让很多 Android 开发者望而却步。例如，音视频开发中很多有特色的或者核心的模块使用 NDK 开发，而 NDK 开发又主要使用 C/C++ 语言编写代码，这对于使用 Java 语言的 Android 开发者来说有门槛。

我为什么要写这本书呢？对于音视频相关技术，网络上遍布零散的知识点，但没有一个成型的知识体系。很多朋友想学习和了解 Android 音视频开发，却不知道如何下手，所以我希望将自己的知识和经验整理成书，帮助读者系统学习、化繁为简，让大家在 Android 音视频开发的道路上不断进步。

本书概要

第 1 章：介绍了音视频基础知识，通过本章学习可以了解一些音视频的基础概念，让读者更好地系统掌握音视频相关知识。

第 2 章：介绍了 Android 应用层使用的系统播放器 MediaPlayer。

第 3 章：介绍了 Android 多媒体管理调度的服务者 MediaPlayerService，以及如何为多媒体播放提供服务。

第 4 章：介绍了 Android 系统中的 StagefrightPlayer。在 Android 系统 5.1 版本之前，其扮演了重要的角色。



Android 音视频开发

第 5 章：介绍了 Android 系统中的 NuPlayer，其是流媒体播放的新生力量。在 Android 系统 5.1 版本之后（包含 5.1 版本），NuPlayer 基于 StagefrightPlayer 的基础类进行构建，利用了更底层的 ALooper/AHandler 机制来异步解码播放。

第 6 章：介绍了 OpenMAX（OMX）框架相关内容。OpenMAX 是一个多媒体应用程序的标准，涉及 OpenMAX IL API 在 Android 应用程序、多媒体框架和编/解码库及其支持的组件（比如 sources 和 sinks）之间建立统一的接口。

第 7 章：介绍了 FFmpeg 库在 Windows、Mac OS 及 Linux 下编译并移植的内容，同时介绍了 FFmpeg 常用的处理音视频的命令。

第 8 章：介绍了 FFmpeg 源码分析及实战开发案例。

第 9 章：介绍了直播技术，主要涉及直播原理、采集数据、编码、推流、播放等。同时提供了一个直播推流完整案例，可以实现一个简单的直播 App。本章还介绍了直播过程中的优化方法，可帮助提升直播体验。

第 10 章：介绍了 H.264 码流结构及 H.265 码流结构。在音视频开发中，可以通过分析数据有无特殊性问题及异常问题来进行排查，帮助定位、修复问题。

第 11 章：介绍了常见的视频封装格式，以及对封装格式的原理和内部结构进行了分析。

读者对象

本书适合具有一定 Android 开发基础并且对音视频技术方向感兴趣的读者阅读，包括：

- 从事 Android 多媒体开发工作的人。
- 从事音视频开发工作的人。
- 从事跨平台 Android 播放器开发工作的人。
- 从 Android 开发想进阶至多媒体、音视频、直播领域的人。
- 从事 Android ROM 开发中维护多媒体播放框架工作的人。
- 对 Android 音视频、播放器、直播技术感兴趣的其他相关人士。

勘误和支持

由于作者的水平有限，书中难免会出现一些错误或者不准确的地方，恳请广大读者批评指正。



另外，我在自己的微信公众号“何俊林”中特意添加了一个新的菜单入口，专门用于展示书中的问题，欢迎读者查看。

如果在阅读本书的过程中，读者有任何疑问或希望和我交流，可以在公众号后台留言或者发邮件到 hejunlin2013@gmail.com，我会一一回复。

致谢

首先要感谢我的家人，谢谢你们在写书期间默默支持着我，还要感谢电子工业出版社博文视点公司付睿老师的耐心校稿，以及感谢同行朋友与我就细节问题进行讨论和对本书的审校。没有你们，就没有本书的诞生，谢谢你们所有人。

读者服务

轻松注册成为博文视点社区用户 (www.broadview.com.cn)，扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34996>





目 录

第 1 章 音视频基础知识	1
1.1 视频编码	1
1.2 音频编码	2
1.3 多媒体播放组件 (Android、iOS)	2
1.4 常见的多媒体框架及解决方案	3
1.5 相关知识点	4
1.5.1 帧率	4
1.5.2 分辨率	4
1.5.3 刷新率	4
1.5.4 编码格式	4
1.5.5 封装格式	4
1.5.6 码率	5
1.5.7 画质与码率	5
1.5.8 DTS 与 PTS	5
1.5.9 YUV 与 RGB	5
1.5.10 视频帧及音频帧	5
1.5.11 量化精度	6
1.5.12 采样率	6
1.5.13 声道	6
第 2 章 常用的系统播放器 MediaPlayer	8
2.1 状态图及生命周期	8
2.2 从创建到 setDataSource 过程	12
2.2.1 从创建到 setDisplay 过程	12
2.2.2 创建过程	13

2.2.3	setDataSource 过程	16
2.2.4	setDisplay 过程	20
2.3	开始 prepare 后的流程	22
2.4	C++中 MediaPlayer 的 C/S 架构	31
第 3 章	管理调度的服务者 MediaPlayerService	40
3.1	Client/Server 通过 IPC 的通信流程图	40
3.2	相关联的类图	42
3.3	产生过程	43
3.4	添加服务的过程	48
3.5	通过 BinderDriver 和 MediaPlayer 通信的过程	50
3.6	创建播放器	55
3.7	建立 StageFright 层交互	58
第 4 章	StagefrightPlayer (AwesomePlayer)	60
4.1	AwsomePlayer 构造过程	60
4.2	AwsomePlayer 使用 MediaExtractor 进行数据解析的过程	66
4.3	AwsomePlayer 解码过程	69
4.3.1	AwsomePlayer 中的 prepare 过程	69
4.3.2	初始化音视频解码器过程	73
4.3.3	使用 OMXCodec 的解码过程	75
4.4	AwsomePlayer 的渲染输出过程	80
4.4.1	用一张图回顾数据处理过程	80
4.4.2	视频渲染器构建过程	81
4.4.3	将音频数据放到 Buffer 的过程	87
4.4.4	AudioPlayer 在 AwsomePlayer 中的运行过程	91
4.4.5	音视频同步	93
4.4.6	音视频输出	96
4.5	概要总结	97
第 5 章	流媒体播放的新生力量 NuPlayer	98
5.1	NuPlayer 整体结构	98
5.2	NuPlayer 的构建过程	100
5.3	NuPlayer 的数据解析模块	102
5.4	NuPlayer 的解码模块	107



5.5 NuPlayer 的渲染模块	109
第 6 章 OpenMAX (OMX) 框架	118
6.1 Codec 部分中的 AwesomePlayer 到 OMX 服务	118
6.1.1 OpenMAX 与 StageFright 框架层级的关系	118
6.1.2 OMX 的初始化流程	120
6.1.3 OMX 中 NodeInstance 列表的管理	127
6.1.4 OMX 中 NodeInstance 节点的操作	127
6.1.5 总结 AwesomePlayer 到 OMX 服务过程	130
6.2 Codec 部分中的 OMXCodec 与 OMX 事件回调流程	131
6.2.1 OMXCodec 与 OMX callback 事件的处理时序图	132
6.2.2 如何从 OMX 中分发事件到 OMXCodec	133
6.2.3 缓冲区更新过程	135
6.2.4 消息回调	137
6.3 MediaCodec 相关知识	139
6.3.1 MediaCodec 的基本认识	139
6.3.2 从创建到 Start 过程	148
6.3.3 MediaCodec 到 OMX 框架过程	154
6.3.4 MediaCodec 硬解码	158
第 7 章 FFmpeg 项目	161
7.1 FFmpeg 简介	161
7.2 在 Windows 下编译 FFmpeg	163
7.2.1 MSYS2	164
7.2.2 Yasm	164
7.2.3 开始编译 FFmpeg-3.1.3	166
7.2.4 创建 shell 编译脚本	167
7.2.5 编译动态库.so	169
7.2.6 编译静态库.a	171
7.3 在 Linux 下编译 FFmpeg	172
7.3.1 在/etc/profile.d 下配置环境变量	172
7.3.2 开始编译 FFmpeg-3.1.3	174
7.3.3 编写 shell 脚本	175
7.3.4 编译动态库.so	176

7.3.5 编译静态库.a	178
7.4 在 Mac OS 下编译 FFmpeg	179
7.4.1 下载源码及配置环境变量	179
7.4.2 开始编译 FFmpeg-3.1.3	183
7.4.3 编写 shell 脚本	183
7.4.4 编译动态库.so	185
7.4.5 编译静态库.a	187
7.5 FFmpeg 常用命令	189
7.5.1 改变帧率、码率和文件大小	189
7.5.2 调整视频分辨率	190
7.5.3 裁剪/填充视频	191
7.5.4 翻转和旋转视频	193
7.5.5 模糊和锐化视频	196
7.5.6 画中画	197
7.5.7 在视频上添加文字	201
7.5.8 文件格式转换	205
7.5.9 时间操作	207
第 8 章 FFmpeg 源码分析及实战	208
8.1 FFmpeg 常用结构体分析	208
8.1.1 AVFormatContext	209
8.1.2 AVInputFormat	211
8.1.3 AVStream	212
8.1.4 AVCodecContext	215
8.1.5 AVPacket	216
8.1.6 AVCodec	218
8.1.7 AVFrame	219
8.1.8 AVIOContext	222
8.1.9 URLProtocol	223
8.1.10 URLContext	224
8.2 FFmpeg 关键函数介绍	225
8.2.1 av_register_all 函数	225
8.2.2 avformat_alloc_context 函数	226
8.2.3 avio_open 函数	226

8.2.4	avformat_open_input 函数	229
8.2.5	avformat_find_stream_info 函数	232
8.2.6	av_read_frame 函数	246
8.2.7	av_write_frame 函数	252
8.2.8	avcodec_decode_video2 函数	256
8.3	FFmpeg 案例（代码实现）	264
8.3.1	利用 FFmpeg 转换格式	264
8.3.2	在实时流中抓取图像	269
8.3.3	在视频中加入水印	277
8.3.4	FFmpeg 音频解码	288
8.3.5	FFmpeg 视频解码	300
8.4	FFPlay 原理	308
8.4.1	注册所有容器格式和 Codec	309
8.4.2	打开流文件	309
8.4.3	读取数据	311
8.4.4	保存数据	318
8.4.5	音视频同步	322
8.4.6	音视频输出	326
第 9 章	直播技术	328
9.1	直播原理	328
9.2	直播架构	328
9.3	直播过程	329
9.3.1	采集数据	329
9.3.2	渲染处理	332
9.3.3	编码数据	333
9.3.4	推流	335
9.3.5	CDN 分发	338
9.3.6	拉流	341
9.3.7	播放流数据	341
9.3.8	直播推流完整案例	343
9.4	流媒体服务器搭建	377
9.5	FFmpeg 推流到流媒体服务器的过程	384
9.6	直播优化那些事	387

9.6.1	卡顿优化	387
9.6.2	延时优化	388
9.6.3	数据代理优化	389
9.6.4	首屏秒开优化	390
9.6.5	弱网优化	391
9.6.6	运营商劫持优化	391
9.6.7	CDN 节点优化	393
第 10 章	H.264 编码及 H.265 编码	395
10.1	H.264 编码框架	395
10.2	H.264 编码原理	395
10.3	H.264 码流分析	397
10.3.1	H.264 编码格式	397
10.3.2	NAL Header	397
10.3.3	H.264 的传输	399
10.3.4	H.264 码流结构	399
10.3.5	H.264 的 Level 和 Profile 说明	406
10.4	H.265 编码框架	408
10.4.1	背景知识	408
10.4.2	H.265 码流结构	409
第 11 章	视频格式分析	414
11.1	MP4 格式分析	414
11.1.1	Box 结构	415
11.1.2	MP4 总体结构	416
11.1.3	movie (moov) box	416
11.1.4	media box	418
11.1.5	sample table (stbl) box	420
11.2	FLV 格式分析	422
11.2.1	FLV 文件结构	422
11.2.2	File Header (文件头)	422
11.2.3	Body	423
11.2.4	Tag	423
11.3	F4V 格式分析	428

11.3.1	file type box	429
11.3.2	movie box	430
11.3.3	movie header box	430
11.3.4	track box	430
11.3.5	media box	431
11.3.6	media information box	433
11.3.7	sample table box	433
11.4	TS 格式分析	437
11.4.1	TS 格式介绍	437
11.4.2	TS 流包含的内容	438
11.4.3	TS 包头解析	438
11.4.4	TS 包传输部分	440
11.4.5	节目专用信息 PSI (Program Specific Information)	440
11.5	AVI 格式分析	444
11.5.1	AVI 整体结构	445
11.5.2	AVI 信息块 ('hdr!' LIST 块)	446
11.5.3	AVI 数据块 ('movi' LIST 块)	447
11.5.4	AVI 索引块 ('idx!'子块)	448
11.6	ASF 格式分析	448
11.6.1	认识 ASF	448
11.6.2	ASF 文件整体结构	449

第 1 章

音视频基础知识

音视频术语是了解音视频开发的基础内容，如一些专有名词、常见的口语化名词等，它们都表述了音视频中客观存在的属性或特征。本章将带领读者了解这些术语以及这些术语背后的含义。

1.1 视频编码

所谓的视频编码就是指通过特定的压缩技术，将某个视频格式文件转换成另一种视频格式文件的方式。视频流传输中最重要的编解码标准有国际电联的 H.261、H.263、H.264，运动静止图像专家组的 M-JPEG 和国际标准化组织运动图像专家组的 MPEG 系列标准，此外在互联网上被广泛应用的还有 Real-Networks 的 RealVideo、微软公司的 WMV 以及 Apple 公司的 QuickTime 等。

视频编码分为两个系列，分别介绍如下。

- MPEG 系列：（由 ISO[国际标准化组织]下属的 MPEG[运动图像专家组]开发）视频编码方面主要是 MPEG1（VCD 用的就是它）、MPEG2（DVD 使用）、MPEG4（DVD RIP 使用的都是它的变种，如 DivX、XviD 等）、MPEG4 AVC（正热门）。其还有音频编码方面，主要是 MPEG Audio Layer 1/2、MPEG Audio Layer 3（大名鼎鼎的 MP3）、MPEG-2 AAC、MPEG-4 AAC 等。注意，DVD 音频没有采用 MPEG 的。
- H.26X 系列：（由 ITU[国际电传视讯联盟]主导，侧重网络传输，注意，只有视频编码）包括 H.261、H.262、H.263、H.263+、H.263++、H.264（就是与 MPEG4 AVC 合作的结晶）。

1.2 音频编码

常见的音频编码格式有 AAC、MP3、AC3，下面分别进行介绍。

- AAC：一种专为声音数据设计的文件压缩格式，与 MP3 不同，它采用了全新的算法进行编码，更加高效，具有更高的“性价比”。利用 AAC 格式，在感觉声音质量没有明显降低的前提下，可使文件更加小巧。苹果 iPod、诺基亚手机也支持 AAC 格式的音频文件。AAC 的优点是，相对于 MP3，AAC 格式的音质更佳，文件更小。AAC 的缺点是，AAC 属于有损压缩格式，与时下流行的 APE、FLAC 等无损压缩格式相比音质存在“本质上”的差距；加之，传输速度更快的 USB 3.0 和 16GB 以上大容量 MP3 正在加速普及，这也使得 AAC 头上“小巧”的光环逐渐暗淡。
- MP3：MP3 是一种音频压缩技术，其全称是动态影像专家压缩标准音频层面 3 (Moving Picture Experts Group Audio Layer III)，简称为 MP3。它被设计用来大幅度地降低音频数据量。利用 MP3 技术，将音乐以 1:10 甚至 1:12 的压缩率，压缩成容量较小的文件，而对于大多数用户来说，重放的音质与最初的不压缩音频相比没有明显下降。MP3 的特点是，其利用人耳对高频声音信号不敏感的特性，将时域波形信号转换成频域信号，并划分成多个频段，对不同的频段使用不同的压缩率，对高频信号使用大压缩率（甚至忽略信号），对低频信号使用小压缩率，保证信号不失真。这样一来就相当于抛弃人耳基本听不到的高频声音，只保留能听到的低频部分，从而将声音用 1:10 甚至 1:12 的压缩率压缩。
- AC3：全称为 Audio Coding Version 3，是 Dolby 实验室所发展的有损音频编码格式。AC3 被广泛应用于 5.1 声道，是 Dolby Pro Logic 的继承者，不同的地方在于 AC3 提供了 6 个独立的声道而 Pro Logic 混合其环绕声道。AC3 普及程度很高，以 384~448kb/s 的码率应用于激光唱片和 DVD，也经常以 640kb/s 的码率广泛应用于电影院。Dolby AC3 提供的环绕声系统由 5 个全频域声道和 1 个超低音声道组成，被称为 5.1 声道。5 个全频域声道包括左前、中央、右前、左后、右后。超低音声道主要提供一些额外的低音信息，使一些场景（如爆炸、撞击等）的声音效果更好。

1.3 多媒体播放组件（Android、iOS）

Android 多媒体播放组件包含 MediaPlayer、MediaCodec、OMX、StageFright、AudioTrack 等，下面分别进行介绍。

- MediaPlayer：播放控制。

- MediaCodec: 音视频编解码。
- OMX: 多媒体部分采用的编解码标准。
- StageFright: 它是一个框架, 替代之前的 OpenCore, 主要做了一个 OMX 层, 仅仅对 OpenCore 的 omx-component 部分做了引用。StageFright 是在 MediaPlayerService 这一层加入的, 和 OpenCore 是并列的。StageFright 在 Android 中是以共享库的形式存在的 (libstagefright.so), 其中的 module——NuPlayer/AwesomePlayer 可用来播放音视频。NuPlayer/AwesomePlayer 提供了许多 API, 可以让上层的应用程序 (Java/JNI) 调用。
- AudiTrack: 音频播放。

iOS 多媒体播放组件包含 VideoToolBox、AudioToolBox、AVPlayer 等, 下面分别进行介绍。

- VideoToolBox: 它是一个底层框架, 提供对硬件编码器和解码器的直接访问。它为视频压缩和解压缩提供服务, 并用于 CoreVideo 像素缓冲区中存储的栅格之间的转换。这些服务是以会话对象的形式 (压缩、解压缩和像素传输), 作为核心基础 (CF) 类型提供的。不需要直接访问硬件编码器和解码器的应用程序都不需要直接使用 VideoToolBox。
- AudioToolBox: 这个框架可以将比较短的声音注册到 System Sound 服务上。注册到 System Sound 服务上的声音被称为 System Sounds。它必须满足下面几个条件。
 - 播放时间不能超过 30s。
 - 数据必须是 PCM 或者 IMA4 流格式的。
 - 必须被打包成下面 3 种格式之一: Core Audio Format (.caf)、Waveform Audio (.wav) 或者 Audio Interchange File (.aiff)。
- AVPlayer: AVPlayer 既可以用来播放音频也可以用来播放视频, 在使用 AVPlayer 的时候, 我们需要导入 AVFoundation.framework 框架, 再引入头文件#import<AVFoundation/AVFoundation.h>。

1.4 常见的多媒体框架及解决方案

常见的多媒体框架及解决方案有 VLC、FFmpeg、GStreamer 等, 下面分别进行介绍。

- VLC: 即 Video LAN Client, 是一款自由、开源的跨平台多媒体播放器及框架。
- FFmpeg: 多媒体解决方案, 不是多媒体框架, 广泛用于音视频开发中。
- GStreamer: 一套构建流媒体应用的开源多媒体框架。

1.5 相关知识点

下面具体介绍音视频相关的名词、术语、概念。

1.5.1 帧率

帧率（Frame Rate）是用于测量显示帧数的量度。所谓的测量单位为每秒显示帧数（frames per second，简称 fps）或“赫兹”（Hz）。

每秒显示帧数（fps）或者帧率表示图形处理器处理场时每秒能够更新的次数。高帧率可以得到更流畅、更逼真的动画。一般来说，30fps 就是可以接受的，但是将性能提升至 60fps 则可以明显提升交互感和逼真感，但是超过 75fps 就不容易察觉有明显的流畅度提升了。如果帧率超过屏幕刷新率，则只会浪费图像处理能力，因为监视器不能以这么快的速度更新，这样超过刷新率的帧率就浪费掉了。

1.5.2 分辨率

视频分辨率是指视频成像产品所形成的图像大小或尺寸。

1.5.3 刷新率

刷新率是指屏幕每秒画面被刷新的次数，刷新率分为垂直刷新率和水平刷新率，一般提到的刷新率通常指垂直刷新率。垂直刷新率表示屏幕上图像每秒重绘多少次，也就是每秒屏幕刷新的次数，以 Hz（赫兹）为单位。刷新率越高，图像就越稳定，图像显示就越自然清晰，对眼睛的影响也越小。刷新率越低，图像闪烁和抖动得就越厉害，眼睛疲劳得就越快。一般来说，如能达到 80Hz 以上的刷新率，就可以完全消除图像闪烁和抖动感，眼睛也不太容易疲劳。

1.5.4 编码格式

编码的目的是压缩数据量，采用编码算法压缩冗余数据。常用的编码格式有如下这两种。

- MPEG（MPEG-2、MPEG-4）
- H.26X（H.263、H.264/AVC、H.265/HEVC）

1.5.5 封装格式

把编码后的音视频数据以一定格式封装到一个容器，封装格式有 MKV、AVI、TS 等。

1.5.6 码率

码率也就是比特率，比特率是单位时间播放连续的媒体（如压缩后的音频或视频）的比特数量。比特率越高，带宽消耗得越多。比特（bit）就是二进制里面最小的单位，要么是 0，要么是 1。

$$\text{文件大小 (b)} = \text{码率 (b/s)} \times \text{时长 (s)}$$

1.5.7 画质与码率

此处提出一个问题，是码率越大，画质越好，视频越流畅吗？这是错误的说法，实际上视频质量和码率、编码算法都有关系。

1.5.8 DTS 与 PTS

下面分别介绍一下 DTS 和 PTS。

- DTS: 即 Decode Time Stamp, 主要用于标示读入内存中的比特流在什么时候开始送入解码器中进行解码。
- PTS: 即 Presentation Time Stamp, 主要用于度量解码后的视频帧什么时候被显示出来。

1.5.9 YUV 与 RGB

下面分别介绍一下颜色空间模型 YUV 与 RGB。

- YUV: 也被称作 YCrCb, 是被欧洲电视系统所采用的一种颜色编码方法（属于 PAL），是 PAL 和 SECAM 模拟彩色电视制式采用的颜色空间模型。其中的 Y、U、V 几个字母不是英文单词的首字母，其中 Y 代表亮度，UV 代表色差，U 和 V 是构成颜色的两个分量。
- RGB: 是一种颜色空间模型，通过对红（R）、绿（G）、蓝（B）3 个颜色通道的变化以及它们相互之间的叠加来得到各式各样的颜色，RGB 即代表红、绿、蓝 3 个通道的颜色。

1.5.10 视频帧及音频帧

常见的视频帧有 I、P、B 帧等，下面分别进行介绍。

- I 帧表示关键帧，你可以理解为这一帧画面的完整保留，解码时只需要本帧数据就可以完成（因为包含完整画面）。

- P 帧表示的是这一帧和之前的一个关键帧（或 P 帧）的差别，解码时需要用之前缓存的画面叠加上本帧定义的差别生成最终画面。（也就是差别帧，P 帧没有完整画面数据，只有与前一帧的画面差别的数据。）
- B 帧是双向差别帧，也就是 B 帧记录的是本帧与前后帧的差别（具体比较复杂，有 4 种情况），换言之，要解码 B 帧，不仅要取得之前的缓存画面，还要解码之后的画面，通过前后画面数据与本帧数据的叠加取得最终的画面。B 帧压缩率高，但是解码时 CPU 会比较吃力。

音频帧的概念没有视频帧那么清晰，几乎所有视频编码格式都可以简单地认为 1 帧就是编码后的一幅图像。但音频帧跟编码格式相关，它是各个编码标准自己实现的。

- 对 PCM（未经编码的音频数据）来说，它根本就不需要帧的概念，根据采样率和采样精度就可以播放。比如采样率为 44.1Hz，采样精度为 16 位的音频，你可以算出比特率是 4 410 016kb/s，每秒的音频数据是固定的 4 410 016/8 字节。
- AMR 帧比较简单，它规定每 20ms 的音频是 1 帧，每一帧音频都是独立的，有可能采用不同的编码算法以及不同的编码参数。
- MP3 帧较复杂一些，包含了更多的信息，比如采样率、比特率等各种参数。具体如下：音频数据帧个数由文件大小和帧长决定，每一帧的长度可能不固定，也可能固定，由比特率决定，每一帧又分为帧头和数据实体两部分，帧头记录了 MP3 的比特率、采样率、版本等信息，每一帧之间相互独立。

1.5.11 量化精度

量化精度表示可以将模拟信号分成多少个等级，量化精度越高，音乐的声压振幅越接近原音乐。量化精度的单位是 bit（比特），CD 标准的量化精度是 16bit，DVD 标准的量化精度是 24bit。也可理解为一个采样点用多少 bit 表示（8/16/24/32bit）。

1.5.12 采样率

采样率指每秒音频采样点个数（8 000/44 100Hz），采样率单位用 Hz（赫兹）表示。

1.5.13 声道

声道（Sound Channel）是指声音在录制或播放时在不同空间位置采集或回放的相互独立的音频信号，所以声道数也就是声音录制时的音源数量或回放时相应的扬声器数量。

常见声道有单声道、立体声道、4 声道、5.1 声道、7.1 声道等，下面分别进行介绍。

- 单声道：设置一个扬声器。
- 立体声道：把单声道一个扬声器扩展为左右对称的两个扬声器。声音在录制过程中被分配到两个独立的声道，从而达到了很好的声音定位效果。这种技术在音乐欣赏中显得尤为有用，听众可以清晰地分辨出各种乐器来自何方，从而使音乐更富想象力，更加接近临场感受。立体声技术广泛应用于自 Sound Blaster Pro 以后的大量声卡，成为了影响深远的音频标准。
- 4 声道：4 声道环绕规定了 4 个发音点，分别是前左、前右、后左、后右，听众则被包围在中间。同时还建议增加一个低音音箱，以加强对低频信号的回放处理（这也就是如今 4.1 声道音箱系统广泛流行的原因）。就整体效果而言，4 声道系统可以为听众带来来自多个不同方向的声音环绕，可以获得身临各种不同环境的听觉感受，给用户以全新的体验。
- 5.1 声道：其实 5.1 声道系统来源于 4.1 声道系统，将环绕声道一分为二，分为左环绕和右环绕，中央位置增加重低音效果，如图 1-1 所示。

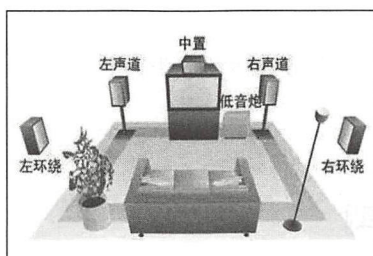


图 1-1 5.1 声道构成图

- 7.1 声道：7.1 声道系统在 5.1 声道系统的基础上又增加了中左和中右两个发音点。简单来说，就是在听者的周围建立起一套前后相对平衡的声场，增加了后中声场声道，如图 1-2 所示。

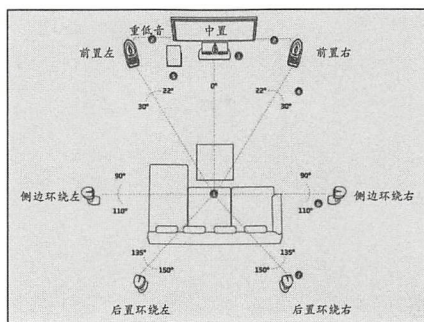


图 1-2 7.1 声道构成图

第 2 章

常用的系统播放器

MediaPlayer

MediaPlayer 是 Android 中的一个多媒体播放类，我们能通过它控制音视频流或本地音视频资源的播放过程。

2.1 状态图及生命周期

MediaPlayer 类用于视频/音频文件的播放控制。本节主要覆盖 MediaPlayer 如下知识点。

- MediaPlayer 的状态图
- Idle 状态
- End 状态
- Error 状态
- Initialized 状态
- Prepared 状态
- Preparing 状态
- Started 状态
- Paused 状态
- Stopped 状态
- PlaybackCompleted 状态

1. MediaPlayer 的状态图

MediaPlayer 用于控制视频/音频文件及流的播放，由状态机进行管理。图 2-1 显示了 MediaPlayer 状态周期。

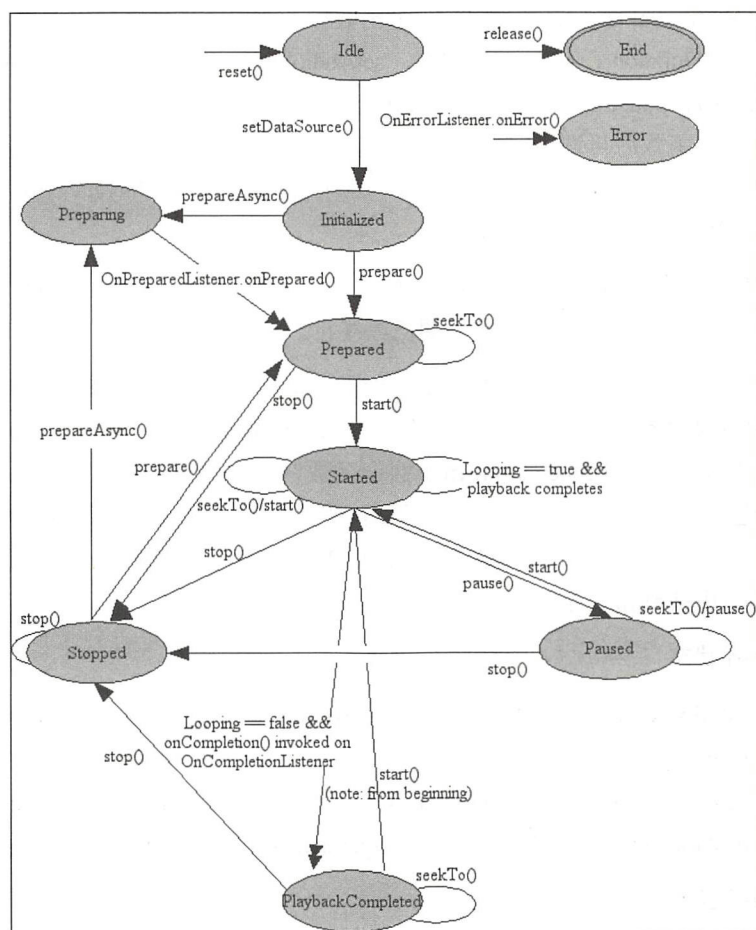


图 2-1 MediaPlayer 状态周期

图 2-1 中的椭圆代表 MediaPlayer 驻留的状态，弧代表播放控制且驱动 MediaPlayer 状态进行过渡。有两种类型的弧，单箭头弧表示的是同步函数调用，双箭头弧表示的是异步函数调用。

从图 2-1 中，我们能看到 MediaPlayer 有下面的一些状态。

2. Idle 状态及 End 状态

在 MediaPlayer 创建实例或者调用 reset 函数后，播放器就被创建了，这时处于 Idle（就

绪) 状态, 调用 `release` 函数后, 就会变成 `End` (结束) 状态, 在这两种状态之间的就是 `MediaPlayer` 的生命周期。

3. Error 状态

在构造一个新 `MediaPlayer` 或者调用 `reset` 函数后, 上层应用程序调用 `getCurrentPosition`、`getVideoHeight`、`getDuration`、`getVideoWidth`、`setAudioStreamType(int)`、`setLooping(boolean)`、`setVolume(float, float)`、`pause`、`start`、`stop`、`seekTo(int)`、`prepare`、`prepareAsync` 这些函数会出错。如果调用 `reset` 函数后再调用它们, 用户提供的回调函数 `OnErrorListener.onError` 将触发 `MediaPlayer` 状态到 `Error` (错误) 状态, 所以一旦不再使用 `MediaPlayer`, 就需要调用 `release` 函数, 以便 `MediaPlayer` 资源得到合理释放。

当 `MediaPlayer` 处于 `End` (结束) 状态时, 它将不能再被使用, 这时不能再回到 `MediaPlayer` 的其他状态, 因为本次生命周期已经终止。

由于支持的音视频格式分辨率过高, 输入数据流超时, 或者其他各种各样的原因将导致播放失败。在这种错误的条件下, 如果用户事先通过 `setOnErrorListener` 注册过 `OnErrorListener`, 当 `player` 内部调用 `OnErrorListener.onError` 回调函数时, 将会返回错误信息。一旦有错误, `MediaPlayer` 会进入 `Error` (错误) 状态, 为了重新使用 `MediaPlayer`, 调用 `reset` 函数, 这时将重新恢复到 `Idle` (就绪) 状态, 所以需要给 `MediaPlayer` 设置错误监听, 出错后就可以从播放器内部返回的信息中找到错误原因。

4. Initialized 状态

当调用 `setDataSource(FileDescriptor)`、`setDataSource(String)`、`setDataSource(Context, Uri)`、`setDataSource(FileDescriptor, long, long)` 其中一个函数时, 将传递 `MediaPlayer` 的 `Idle` 状态变成 `Initialized` (初始化) 状态, 如果 `setDataSource` 在非 `Idle` 状态时调用, 会抛出 `IllegalStateException` 异常。当重载 `setDataSource` 时, 需要抛出 `IllegalArgumentException` 和 `IOException` 这两个异常。

5. Prepared 状态

`MediaPlayer` 有两种途径到达 `Prepared` 状态, 一种是同步方式, 另一种是异步方式。同步方式主要使用本地音视频文件, 异步方式主要使用网络数据, 需要缓冲数据。调用 `prepare` (同步函数) 将传递 `MediaPlayer` 的 `Initialized` 状态变成 `Prepared` 状态, 或者调用 `prepareAsync` (异步函数) 将传递 `MediaPlayer` 的 `Initialized` 状态变成 `Preparing` 状态, 最后到 `Prepared` 状态。如果应用层事先注册过 `setOnPreparedListener`, 播放器内部将回调用户设置的 `OnPreparedListener` 中的 `onPrepared` 回调函数。注意, `Preparing` 是一个瞬间状态 (可理解为时间比较短)。

6. Started 状态

在 MediaPlayer 进入 Prepared 状态后，上层应用即可设置一些属性，如音视频的音量、screenOnWhilePlaying、looping 等。在播放控制开始之前，必须调用 start 函数并成功返回，MediaPlayer 的状态开始由 Prepared 状态变成 Started 状态。当处于 Started 状态时，如果用户事先注册过 setOnBufferingUpdateListener，播放器内部会开始回调 OnBufferingUpdateListener.onBufferingUpdate，这个回调函数主要使应用程序保持跟踪音视频流的 buffering（缓冲）status，如果 MediaPlayer 已经处于 Started 状态，再调用 start 函数是没有任何作用的。

7. Paused 状态

MediaPlayer 在播放控制时可以是 Paused（暂停）和 Stopped（停止）状态的，且当前的播放时进度可以被调整，当调用 MediaPlayer.pause 函数时，MediaPlayer 开始由 Started 状态变成 Paused 状态，这个从 Started 状态到 Paused 状态的过程是瞬间的，反之在播放器内部是异步过程的。在状态更新并调用 isPlaying 函数前，将有一些耗时。已经缓冲过的数据流，也要耗费数秒。

当 start 函数从 Paused 状态恢复回来时，playback 恢复之前暂停时的位置，接着开始播放，这时 MediaPlayer 的 Paused 状态又变成 Started 状态。如果 MediaPlayer 已经处于 Paused 状态，这时再调用 pause 函数是没有任何作用的，将保持 Paused 状态。

8. Stopped 状态

当调用 stop 函数时，MediaPlayer 无论正处于 Started、Paused、Prepared 或 PlaybackCompleted 中的哪种状态，都将进入 Stopped 状态。一旦处于 Stopped 状态，playback 将不能开始，直到重新调用 prepare 或 prepareAsync 函数，且处于 Prepared 状态时才可以开始。

如果 MediaPlayer 已经处于 Stopped 状态了，这时再调用 stop 函数是没有任何作用的，将保持 Stopped 状态。

在 Seek 操作完成后，如果事先在 MediaPlayer 注册了 setOnSeekCompleteListener，播放器内部将回调 OnSeekComplete.onSeekComplete 函数。当然 seekTo 函数也可以在其他状态下被调用，如 Prepared、Paused 及 PlaybackCompleted 状态。

9. PlaybackCompleted 状态

当前播放的位置可以通过 getCurrentPosition 函数获取，通过 getCurrentPosition 函数，可以跟踪播放器的播放进度。

当 MediaPlayer 播放到数据流的末尾时，一次播放过程完成。在 MediaPlayer 中事先调用 setLooping(boolean)并设置为 true，表示循环播放，MediaPlayer 依然处于 Started 状态。如果调

用 `setLooping(boolean)` 并设置为 `false`（表示不循环播放），并且事先在 `MediaPlayer` 上注册过 `setOnCompletionListener`，播放器内部将回调 `OnCompletion.onCompletion` 函数，这就表明 `MediaPlayer` 开始进入 `PlaybackCompleted`（播放完成）状态。当处于 `PlaybackCompleted` 状态时，调用 `start` 函数，将重启播放器从头开始播放数据。

2.2 从创建到 `setDataSource` 过程

本节分析的是从 `MediaPlayer` 创建到 `MediaPlayer` 调用 `setDataSource` 的过程。

在以前的相关图书中总是把时序图放在最后用于总结，但这样会让人觉得开始时没有一个概览，无从下手，所以本节先附上时序图，一步一步地对着时序图介绍每个阶段。

2.2.1 从创建到 `setDisplay` 过程

`MediaPlayer` 时序图一（`create` 到 `setDataSource` 过程，后面章节还有时序图，故这么命名），如图 2-2 所示。

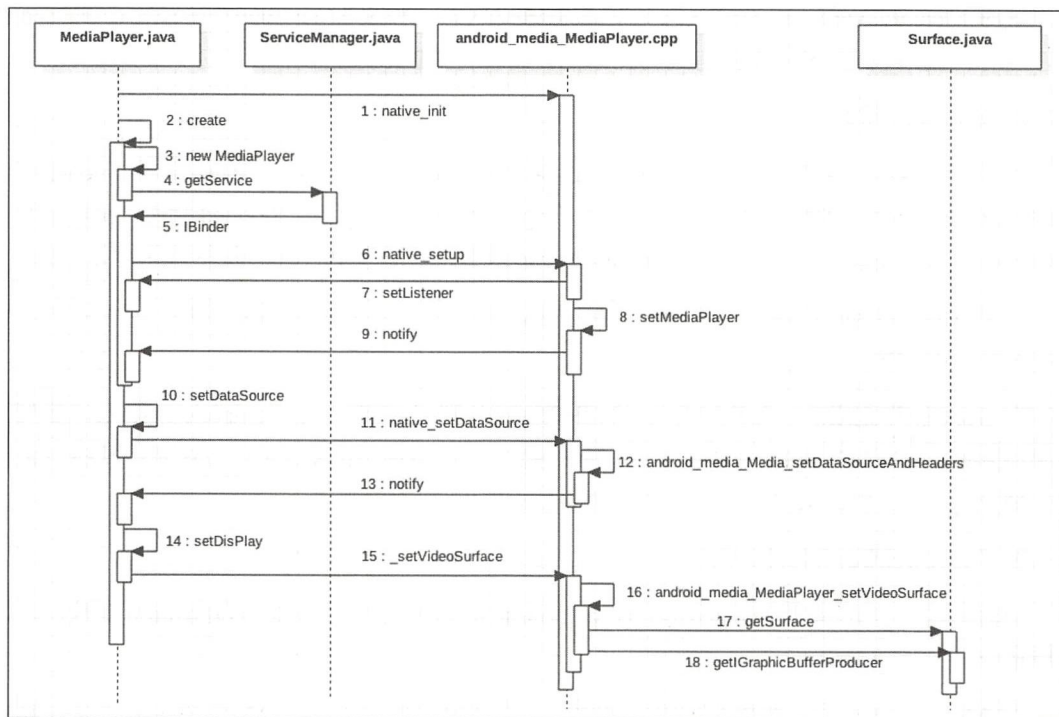


图 2-2 从 `create` 到 `setDisplay` 过程的时序图

从时序图可以看到，通过 `getService` 从 `ServiceManager` 获取对应的 `MediaPlayerService`，然后调用 `native_setup` 函数创建播放器，接着调用 `setDataSource` 把 URL 地址传入底层。当准备好后，通过 `setDisplay` 传入 `SurfaceHolder`，以便将解码出的数据放到 `SurfaceHolder` 中的 `Surface`。最后显示在 `SurfaceView` 上。

2.2.2 创建过程

当外部调用 `MediaPlayer.create(this, "http://www.xxx.mp4")` 时，进入 `MediaPlayer` 的创建过程：

```
public static MediaPlayer create(Context context, Uri uri, SurfaceHolder
holder, AudioAttributes audioAttributes, int audioSessionId) {
    try {
        MediaPlayer mp = new MediaPlayer();
        final AudioAttributes aa = audioAttributes != null ?
audioAttributes : new AudioAttributes.Builder().build();
        //声音相关处理，若为空，就创建一个
        mp.setAudioAttributes(aa); //设置音频属性
        mp.setAudioSessionId(audioSessionId);
        //设置声音的会话 ID，视频和音频是分开渲染的
        mp.setDataSource(context, uri);
        //从这里开始 setDataSource。uri 即统一资源标识符
        if (holder != null) { //判断 SurfaceHolder 是否为空，这是一个控制器，
//Surface 的控制器，用来操纵 Surface，处理它在 Canvas 上作画的效果和动画，
//控制表面、大小、像素等
            mp.setDisplay(holder); //给 Surface 设置一个 Surface 的控制器
        }
        mp.prepare(); //开始准备
        return mp;
    } catch (IOException ex) {
        //省略各种 try、catch 操作
    }
    return null;
}
```

以上代码可以总结为，当 `MediaPlayer` 通过 `create` 的方式创建播放器时，内部 `new` 出 `MediaPlayer` 对象，并 `setDataSource`，做好 `prepare` 的动作。这时外部只需调用 `start` 函数，就能播放音视频资源了。

实例化 `MediaPlayer` 有如下两种方式。

(1) 可以使用直接 `new` 的方式：

```
MediaPlayer mp = new MediaPlayer();
```

Android 音视频开发

(2) 也可以使用 `create` 的方式，如：

```
MediaPlayer mp = MediaPlayer.create(this, R.raw.test);
//这时就不用调用 setDataSource 了
```

上面两种实例化 `MediaPlayer` 的方式，都要经过 `new MediaPlayer`，下面看看构造中做了什么操作。

播放页处理主页面：

```
public MediaPlayer() {
    Looper looper; //定义一个Looper
    if ((looper = Looper.myLooper()) != null) {
        //如果 myLooper 不为空就赋值到 looper
        mEventHandler = new EventHandler(this, looper);
        //实例化一个 EventHandler 对象
    } else if ((looper = Looper.getMainLooper()) != null) {
        //或者如果主线程 Looper 不为空，也可以赋值到 looper
        mEventHandler = new EventHandler(this, looper);
    } else {
        mEventHandler = null;
    }
    mTimeProvider = new TimeProvider(this);
    //时间数据容器，一般 provider 都是和数据联系起来的，如 VideoProvider、
    //ContentProvider
    mOpenSubtitleSources = new Vector<InputStream>();
    //通过 Binder 机制获取系统原生服务，像 Camera、MediaRecorder、
    //MediaPlayer 等服务都会申请，用于打开摄像头、获取声音等
    IBinder b = ServiceManager.getService(Context.APP_OPS_SERVICE);
    mAppOps = IAppOpsService.Stub.asInterface(b); //Binder 的服务连接
    //桥，把 IPC 通信过程中的相关服务、对象带回，下面就能使用 native_setup 开始创建
    //MediaPlayer 了。此处为软引用，意为用 C++更好地创建 MediaPlayer
    native_setup(new WeakReference<MediaPlayer>(this));
}
```

接下来看 Native 层如何创建一个 `MediaPlayer`。在介绍 `native_setup` 之前，请注意一般都是静态代码块中加载 `.so` 文件的，在 `MediaPlayer` 中有一段静态代码块，用于加载和链接库文件 `media_jni.so`，早于构造函数，在加载类时就执行。一般全局性的数据、变量都可以放在这里。下面是加载和链接 `media_jni.so` 文件的代码：

```
static {
    System.loadLibrary("media_jni"); //media_jni.so
    native_init();
}
```


下面开始进入 android_media_MediaPlayer.cpp 分析，第一个函数 android_media_MediaPlayer_native_init 就是从 Java 静态代码块调过来的 native_init:

```
static void
android_media_MediaPlayer_native_init(JNIEnv *env)
//可理解为一个万能指针表，通过操作符 (->) 访问 JNI 中的函数
{
    jclass clazz; //类的句柄
    clazz = env->FindClass("android/media/MediaPlayer");
    //这里通过 Native 层调用 Java 层，获取 MediaPlayer 对象
    if (clazz == NULL) { //找不到，直接 return
        return;
    }
    fields.context = env->GetFieldID(clazz, "mNativeContext", "J");
    //获取成员变量 mNativeContext，它在 MediaPlayer.java 中是一个 long 型整数，实际对应的是一个内存地址
    if (fields.context == NULL) {
        return;
    }
    fields.post_event = env->GetStaticMethodID(clazz,
"postEventFromNative", "(Ljava/lang/Object;IIILjava/lang/Object;)V");
    if (fields.post_event == NULL) {
        return;
    }
    //省略部分代码
}
```

上面这种方式是通过 JNI 调用 Java 层的 MediaPlayer 类，然后拿到 mNativeContext 的指针，接着调用了 MediaPlayer.java 中的静态方法 postEventFromNative，把 Native 的事件回调到 Java 层，使用 EventHandler post 事件回到主线程中，用软引用指向原生的 MediaPlayer，以保证 Native 代码是安全的。代码如下：

```
private static void postEventFromNative(Object mediaplayer_ref,
    int what, int arg1, int arg2, Object obj){
    MediaPlayer mp = (MediaPlayer)((WeakReference)mediaplayer_ref).
get(); //得到软引用对象
    if (mp == null) {
        return;
    }
    //省略部分代码
    if (mp.mEventHandler != null) { //handler 不为空，发送一条 msg
        Message m = mp.mEventHandler.obtainMessage(what, arg1, arg2, obj);
        mp.mEventHandler.sendMessage(m);
    }
}
```


Android 音视频开发

```
}
```

之前我们在 Java 层的 MediaPlayer.java 文件的构造函数中，分析到最后有一个 native_setup，在 android_media_MediaPlayer.cpp 中找到对应的函数，代码如下：

```
static void android_media_MediaPlayer_native_setup (JNIEnv *env, jobject
thiz, jobject weak_this)
{
    sp<MediaPlayer> mp = new MediaPlayer();
    //省略部分代码
    //给 MediaPlayer 创建一个 listener，以便我们在 Java 层设置的
    //setPrepareListener、setOnCompleteListener 能产生回调
    sp<JNIMediaPlayerListener> listener = new JNIMediaPlayerListener(env,
thiz, weak_this);
    mp->setListener(listener);
    //对于 Java 层来说，C++中的 MediaPlayer 是不透明的，也无须关心其对应的逻辑，各司
    //其职即可
    setMediaPlayer(env, thiz, mp);
}
```

可以看到会设置一些回调用的 listener 及创建 C++中的 MediaPlayer 对象。

2.2.3 setDataSource 过程

上面就是 MediaPlayer 的构造过程。构造后接下来要设置数据源，进而到了 setDataSource 过程，下面看看 setDataSource 做了什么操作：

```
public void setDataSource(String path)
    throws IOException, IllegalArgumentException, SecurityException,
IllegalStateException {
    setDataSource(path, null, null);
} //setDataSource 包含文件路径或 HTTP/RTSP 地址
public void setDataSource(String path, Map<String, String> headers) throws
IOException, IllegalArgumentException, SecurityException, IllegalStateException
{
    String[] keys = null;
    String[] values = null;
    if (headers != null) {
        keys = new String[headers.size()]; //请求头对应的 key
        values = new String[headers.size()]; //请求头对应的 value
        int i = 0;
        //把 HTTP/RTSP 中包含的 key、value 分别装到两个数组中
        for (Map.Entry<String, String> entry: headers.entrySet()) {
            keys[i] = entry.getKey();
```

第2章 常用的系统播放器 MediaPlayer

```

        values[i] = entry.getValue();
        ++i;
    }
}
//送至 setDataSource
setDataSource(path, keys, values);
}
private void setDataSource(String path, String[] keys, String[] values) throws
IOException, IllegalArgumentException, SecurityException, IllegalStateException {
    final Uri uri = Uri.parse(path); //解析 path
    final String scheme = uri.getScheme();
    if ("file".equals(scheme)) {
        path = uri.getPath();
    } else if (scheme != null) {
        //1.处理非文件资源
        nativeSetDataSource(
            MediaHTTPService.createHttpServiceBinderIfNecessary(path),
            path, keys, values);
        return;
    }
    //2.处理文件类型
    final File file = new File(path);
    if (file.exists()) {
        FileInputStream is = new FileInputStream(file);
        FileDescriptor fd = is.getFD(); //得到文件标识符
        setDataSource(fd);
        is.close();
    } else {
        throw new IOException("setDataSource failed.");
    }
}
}

```

先看看 `setDataSource` 中传入的参数是文件描述符的情况：

```

public void setDataSource(FileDescriptor fd )
    throws IOException, IllegalArgumentException, IllegalStateException
{
    setDataSource(fd, 0, 0x7fffffffffffffffL);
}
public void setDataSource(FileDescriptor fd, long offset, long length)
    throws IOException, IllegalArgumentException, IllegalStateException
{
    _setDataSource(fd, offset, length);
}
private native void _setDataSource(FileDescriptor fd, long offset, long

```


Android 音视频开发

```
length)
    throws IOException, IllegalArgumentException, IllegalStateException;
```

开始进入 JNI 层，发现找不到 `android_media_MediaPlayer_setDataSource` 函数，但发现有一个函数名映射函数声明，这是 JNI 中常用的动态注册方法，代码如下：

```
static JNINativeMethod gMethods[] = {
{
    "nativeSetDataSource",
    "(Landroid/os/IBinder;Ljava/lang/String;[Ljava/lang/String;"
    "[Ljava/lang/String;)V",
    (void *)android_media_MediaPlayer_setDataSourceAndHeaders
},
{"_setDataSource",      "(Ljava/io/FileDescriptor;JJ)V",
 (void *)android_media_MediaPlayer_setDataSourceFD},
{"_setDataSource",      "(Landroid/media/MediaDataSource;)V",
 (void *)android_media_MediaPlayer_setDataSourceCallback },
{"_setVideoSurface",    "(Landroid/view/Surface;)V",
 (void *)android_media_MediaPlayer_setVideoSurface},
//省略相关函数名映射 native 函数声明
};
```

对以上这个函数名映射，如果读者看过 `JNINativeMethod` 源码的话，应该不会感到陌生，无非还是映射，不影响我们的分析。在这里接下来对 `android_media_MediaPlayer_setDataSourceFD` 函数进行分析：

```
static void
android_media_MediaPlayer_setDataSourceFD(JNIEnv *env, jobject thiz,
jobject fileDescriptor, jlong offset, jlong length)
{
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz); //得到 MediaPlayer 对象
    //省略抛出异常代码
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    //在 JNI 中获取 java.io.FileDescriptor
    ALOGV("setDataSourceFD: fd %d", fd);
    process_media_player_call( env, thiz, mp->setDataSource(fd, offset,
length), "java/io/IOException", "setDataSourceFD failed." );
}
//这里开始调用 JNIEnv*中的 GetIntField 函数获取对应的变量
int jniGetFDFromFileDescriptor(JNIEnv* env, jobject fileDescriptor) {
    return (*env)->GetIntField(env, fileDescriptor, gCachedFields.
descriptorField);
}
```

接着分析 `process_media_player_call` 函数：

第2章 常用的系统播放器 MediaPlayer

```

static void process_media_player_call(JNIEnv *env, jobject thiz, status_t
opStatus, const char* exception, const char *message)
{
    if (exception == NULL) { //不抛出异常, 发送一个 onError 事件
        if (opStatus != (status_t) OK) {
            //如果在 setDataSource 过程中 opStatus 不 ok
            sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
            if (mp != 0) mp->notify(MEDIA_ERROR, opStatus, 0);
            //通知 MEDIA_ERROR
        }
    }
    //省略抛出异常部分代码
}

```

总结以上代码: 当 mp->setDataSource(fd, offset, length)函数得到状态后, 对各种状态进行通知。有异常的直接抛出, 这样也就不会影响 MediaPlayer 后面的执行过程了。

接下来看看以 HTTP/RTSP 传入 JNI。在 Java 层中对应的 nativeSetDataSource 函数如下:

```

private native void nativeSetDataSource(IBinder httpServiceBinder, String
path, String[] keys, String[] values)throws IOException,
IllegalArgumentException, SecurityException, IllegalStateException;

```

在 JNI 中通过映射表可对应到 android_media_MediaPlayer_setDataSourceAndHeaders 函数:

```

static void
android_media_MediaPlayer_setDataSourceAndHeaders(
    JNIEnv *env, jobject thiz, jobject httpServiceBinderObj, jstring
path, jobjectArray keys, jobjectArray values) {
    //获取 MediaPlayer 对象
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    //省略判空, 抛出异常代码
    const char *tmp = env->GetStringUTFChars(path, NULL);
    if (tmp == NULL) { //内存溢出
        return;
    }
    ALOGV("setDataSource: path %s", tmp);
    String8 pathStr(tmp);
    env->ReleaseStringUTFChars(path, tmp);
    tmp = NULL;
    //省略部分代码
    sp<IMediaHTTPService> httpService;
    if (httpServiceBinderObj != NULL) {
        sp<IBinder> binder = ibinderForJavaObject(env, httpServiceBinderObj);
        //通过 Binder 机制将 httpServiceBinderObj 传给 IPC 并返回给 binder
        //然后强制转换成 IMediaHTTPService
    }
}

```


Android 音视频开发

```

        httpService = interface_cast<IMediaHTTPService>(binder);
    }

    //开始判断状态, 和上面的文件操作是一样的
    status_t opStatus =
        mp->setDataSource(
            httpService,
            pathStr,
            headersVector.size() > 0? &headersVector : NULL);
    //见上面的文件操作
    process_media_player_call(
        env, this, opStatus, "java/io/IOException",
        "setDataSource failed." );
}

```

至此, `setDataSource` 过程就完成了。这里需要注意两点, 一点是从 `Java→JNI→C++` 的正向调用过程 (前面从 `Java` 层到 `Native` 层都是正向过程), 一点是 `C++→JNI→Java` 的过程 (如 `mp->setDataSource(httpService, pathStr, headersVector.size() > 0? &headersVector : NULL)`), 那有读者肯定会问, 这样来回调的好处是什么? 好处有如下这几点。

- 安全性, 封装在 `Native` 层的代码是 `so` 形式的, 破坏性风险小。
- 效率高, 在运行速度上 `C++` 执行时间短, 且底层也是用 `C++` 语言编写的。对于复杂的渲染及对时间要求高的渲染, 放在 `Native` 层是最好不过的选择。
- 连通性, 正向调用将值传入, 反向调用把处理过的值通知回去。相当于一根管道。

2.2.4 setDisplay 过程

接下来看看在 `setDataSource` 之后, 开始进行的 `mp.setDisplay(holder)`:

```

public void setDisplay(SurfaceHolder sh) {
    mSurfaceHolder = sh; //1. 给 Surface 设置一个控制器, 用于展示视频图像
    Surface surface;
    if (sh != null) {
        surface = sh.getSurface();
    } else {
        surface = null;
    }
    _setVideoSurface(surface); //2. 给视频设置 Surface, 带_的函数是 native 函数
    updateSurfaceScreenOn(); //3. 更新 Surface 到屏幕上
}

```

对于上面代码中的第 2 点, 同样在 `android_media_MediaPlayer.cpp` 中找到其对应的函数:

```
static void
```

第2章 常用的系统播放器 MediaPlayer

```

    android_media_MediaPlayer_setVideoSurface(JNIEnv *env, jobject thiz,
jobject jsurface)
    {
        setVideoSurface(env, thiz, jsurface, true /* mediaPlayerMustBeAlive
*/);
    }

    static void
    setVideoSurface(JNIEnv *env, jobject thiz, jobject jsurface, jboolean
mediaPlayerMustBeAlive)
    {
        sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
        //省略抛出异常的代码
        decVideoSurfaceRef(env, thiz);
        //IGraphicBufferProducer
        sp<IGraphicBufferProducer> new_st;
        if (jsurface) {
            //得到 Java 层的 Surface
            sp<Surface> surface(android_view_Surface_getSurface(env, jsurface));
            if (surface != NULL) { //不为空, 获取 IGraphicBufferProducer
                new_st = surface->getIGraphicBufferProducer();
                //省略抛出异常的代码
                //调用 incStrong
                new_st->incStrong((void*)decVideoSurfaceRef);
            } else {
                //省略抛出异常的代码
                return;
            }
        }
        env->SetLongField(thiz, fields.surface_texture, (jlong)new_st.get());
        //如果 MediaPlayer 还未被初始化, setDataSource 将失败, 但 setDataSource 之前就
//setDisplay 了, 在 prepare/prepareAsync 中调用 setVideoSurfaceTexture 可以覆盖该 case
        mp->setVideoSurfaceTexture(new_st);
    }
    static void
    decVideoSurfaceRef(JNIEnv *env, jobject thiz)
    {
        sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
        //省略部分代码
        //得到旧的 SurfaceTexture
        sp<IGraphicBufferProducer> old_st = getVideoSurfaceTexture(env, thiz);
        if (old_st != NULL) {
            old_st->decStrong((void*)decVideoSurfaceRef);
        }
    }
}

```




这里有如下几个概念需要理解。

- **SurfaceTexture**: SurfaceTexture 是 Android 3.0 (API 11) 加入的一个类。这个类跟 SurfaceView 很像, 可以从视频解码里面获取图像流 (image stream)。但是, 和 SurfaceView 不同的是, SurfaceTexture 在接收图像流之后, 不需要显示出来。SurfaceTexture 不需要显示到屏幕上, 因此我们可以用 SurfaceTexture 接收解码出来的图像流, 然后从 SurfaceTexture 中取得图像帧的副本进行处理, 处理完后再送给另一个 SurfaceView 用于显示。
- **Surface**: 处理被屏幕排序的原生的 Buffer, Android 中的 Surface 就是一个用来画图形 (graphic) 或图像 (image) 的地方。对于 View 及其子类, 都是画在 Surface 上的, 各 Surface 对象通过 SurfaceFlinger 合成到 frameBuffer。每个 Surface 都是双缓冲的 (实际上就是两个线程, 一个渲染线程, 一个 UI 更新线程), 它有一个 backBuffer 和一个 frontBuffer。在 Surface 中创建的 Canvas 对象, 可用来管理 Surface 绘图操作, Canvas 对应 Bitmap, 存储 Surface 中的内容。
- **SurfaceView**: 在 Camera、MediaRecorder、MediaPlayer 中 SurfaceView 经常被用来显示图像。SurfaceView 是 View 的子类, 实现了 Parcelable 接口, 其中内嵌了一个专门用于绘制的 Surface, SurfaceView 可以控制这个 Surface 的格式和尺寸, 以及 Surface 的绘制位置。可以理解 Surface 就是管理数据的地方, SurfaceView 就是展示数据的地方。
- **SurfaceHolder**: 顾名思义, 是一个管理 SurfaceHolder 的容器。SurfaceHolder 是一个接口, 其可被理解为一个 Surface 的监听器。通过回调函数 addCallback(SurfaceHolder.Callback callback) 监听 Surface 的创建, 通过获取 Surface 中的 Canvas 对象, 锁定之。所得到的 Canvas 对象在完成修改 Surface 中的数据后, 释放同步锁, 并提交改变 Surface 的状态及图像, 展示新的图像数据。

最后总结一下, SurfaceView 中调用 getHolder 函数, 可以获得当前 SurfaceView 中的 Surface 对应的 SurfaceHolder, SurfaceHolder 开始对 Surface 进行管理操作。这里按 MVC 模式可以更好地理解 M:Surface (图像数据)、V:SurfaceView (图像展示)、C:SurfaceHolder (图像数据管理)。MediaPlayer.java 中的 setDisplay 操作就是对将要显示的视频进行预设置。

以上就是 setDisplay 的过程, Java 层中 setDisplay 的最后一行, 就是通过 JNI 返回的 Surface, 时时做好更新准备。

2.3 开始 prepare 后的流程

在 2.2 节中分析了 MediaPlayer 从创建到 setDataSource 的过程, 尽管分析了代码, 但是没



有从 MediaPlayer 生态上认识各类库之间的依赖调用关系。

MediaPlayer 部分的头文件在 frameworks/base/include/media/ 目录中，这个目录和 libmedia.so 库源文件的目录 frameworks/base/media/libmedia/ 相对应。主要的头文件有以下几个。

- IMediaPlayerClient.h
- mediaplayer.h
- IMediaPlayer.h
- IMediaPlayerService.h
- MediaPlayerInterface.h

在这些头文件中，mediaplayer.h 提供了对上层的接口，而其他的几个头文件提供的是一些接口类（即包含了纯虚函数的类），这些接口类必须被实现类继承才能够使用。

MediaPlayer 各个具体类之间的依赖关系图如图 2-3 所示。

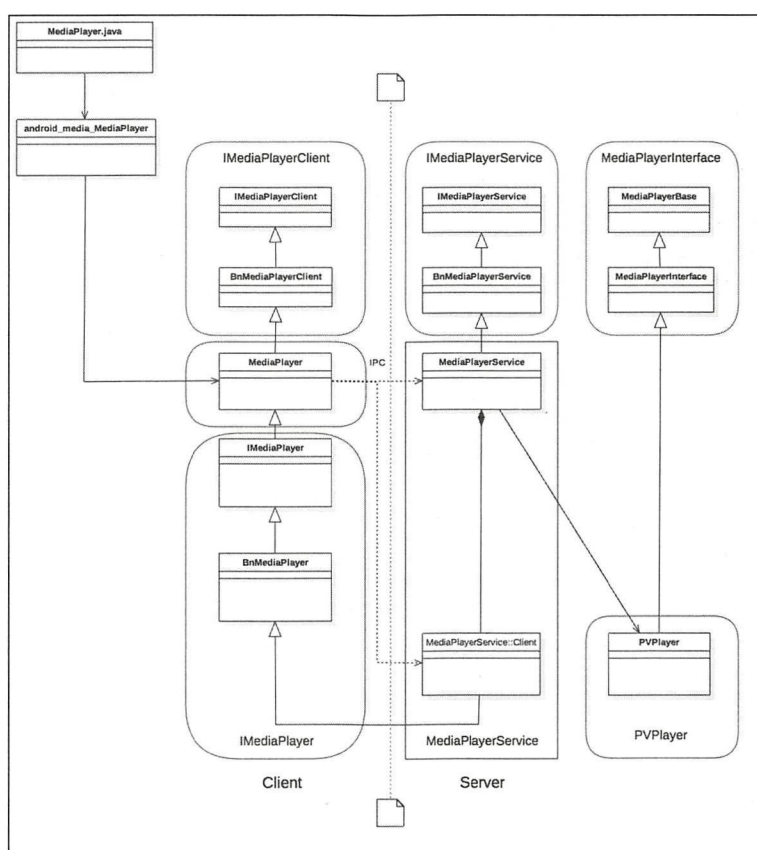


图 2-3 MediaPlayer 各个具体类之间的依赖关系图



Android 音视频开发

在运行的时候，整个 MediaPlayer 可以大致上分成 Client 和 Server 两个部分，它们分别在两个进程中运行，它们之间使用 Binder 机制实现 IPC 通信。从框架结构上看，IMediaPlayerService.h、IMediaPlayerClient.h 和 mediaplayer.h 这 3 个头文件中定义了 MediaPlayer 的接口和架构，在目录中有专门的 MediaPlayerService.cpp 和 mediaplayer.cpp 文件对应上面 3 个头文件，用于 MediaPlayer 架构的实现。

在给播放器设置数据源且展现了 Surface 后，你应当开始调用 prepare 或 prepareAsync 函数。对于文件类型，调用 prepare 函数将暂时阻塞，因为 prepare 是一个同步函数，直到 MediaPlayer 已经准备好数据即将播放，也就是播放器回调了 onPrepared 函数，进入 Prepared 状态。

prepare 函数的执行过程如下：

```
public void prepare() throws IOException, IllegalStateException {
    _prepare(); //调用 native 函数
    scanInternalSubtitleTracks();
}
```

Native 层的 android_media_MediaPlayer_prepare 函数：

```
static void android_media_MediaPlayer_prepare(JNIEnv *env, jobject thiz)
{
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz); //获取 MediaPlayer 对象
    if (mp == NULL) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }
    sp<IGraphicBufferProducer> st = getVideoSurfaceTexture(env, thiz); // 1
    mp->setVideoSurfaceTexture(st); // 2
    process_media_player_call( env, thiz, mp->prepare(), "java/io/IOException",
    "Prepare failed." ); // 3
}

static sp<IGraphicBufferProducer>
getVideoSurfaceTexture(JNIEnv* env, jobject thiz) {
    IGraphicBufferProducer * const p = (IGraphicBufferProducer*)env->
    GetLongField(thiz, fields.surface_texture);
    return sp<IGraphicBufferProducer>(p);
}
```

我们曾经介绍过上面代码中的 1、2、3，1 中的 getVideoSurfaceTexture 获取一个 IGraphicBufferProducer 类型指针，2 中是 setVideoSurfaceTexture，主要给 MediaPlayer 传入 IGraphicBufferProducer。这里 IGraphicBufferProducer 就是 App 和 BufferQueue 的重要桥梁，GraphicBufferProducer 承担着单个应用进程中的 UI 显示需求，与 BufferQueue 打交道的就是它。



BpGraphicBufferProducer 是 GraphicBufferProducer 在客户端这边的代理对象，负责和 SurfaceFlinger 交互，GraphicBufferProducer 通过 gbp（IGraphicBufferProducer 类对象）向 BufferQueue 获取 Buffer，然后填充 UI 信息，填充完毕会通知 SurfaceFlinger。

3 中是一个判定并且进行通知的函数，这个 process_media_player_call 是对 MediaPlayer 调用 prepare 函数后是否有异常的检测，如果出现参数不合法，或是 I/O 异常，就会抛出异常。

我们知道 MediaPlayer 还有一个 prepareAsync 函数，前面的思路都是顺着 MediaPlayer 中的 create 函数来的。

如果是下面这种场景，即一个网络 URL 被发送过来，就是网络流数据传入 MediaPlayer，这时就要用到 prepareAsync 函数了：

```
public void startPlayUrl(Uri uri) {
    MediaPlayer mMediaPlayer = new MediaPlayer();
    try{
        mMediaPlayer.setDataSource(this, uri);
    } catch (Exception e) { //这里本来有 4 个异常，此处为了举例，使用了一个大
//Exception 替代，实际开发中还是要使用那 4 个 Exception 函数
        e.printStackTrace();
    }
    mMediaPlayer.setOnPreparedListener(mPreparedListener);
    mMediaPlayer.setOnVideoSizeChangedListener (mVideoSizeChangedListener);
    mMediaPlayer.setOnErrorListener(mOnErrorListener);
    mMediaPlayer.prepareAsync();
}
```

分析 MediaPlayer 中的 prepareAsync 函数：在 setDataSource 中且展现了 Surface 后，开始调用 prepare 或 prepareAsync 函数，对于网络视频流类型，尽量调用 prepareAsync 函数，因为是异步的，不会导致没有足够的数据影响起播，代码如下：

```
public native void prepareAsync() throws IllegalStateException;
```

这是一个 native 函数声明，下面分析 android_media_MediaPlayer_prepareAsync 函数：

```
static void android_media_MediaPlayer_prepareAsync(JNIEnv *env, jobject
thiz)
{
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    if (mp == NULL) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }
    sp<IGraphicBufferProducer> st = getVideoSurfaceTexture(env, thiz);
    mp->setVideoSurfaceTexture(st);
}
```




```
process_media_player_call( env, this, mp->prepareAsync(), "java/io/
IOException", "Prepare Async failed." );
}
```

从代码来分析，除了最后 `process_media_player_call` 中的 `mp->prepareAsync` 在判断状态时不一样，其他都和 `prepare` 函数是一样的。它的操作结果经过回调通知 Java 层。

下面看看 `media/mediaplayer.h` 中的 `prepareAsync` 函数（C++代码）：

```
status_t MediaPlayer::prepareAsync() //之前所说的 status_t 就是从这里返回的
{
    ALOGV("prepareAsync");
    Mutex::Autolock _l(mLock); //这是一个互斥锁
    return prepareAsync_l();
}
```

接着分析 `prepareAsync_l` 函数实现代码：

```
//必须在锁住后调用此函数
status_t MediaPlayer::prepareAsync_l()
{
    if ( (mPlayer != 0) && ( mCurrentState & ( MEDIA_PLAYER_INITIALIZED
| MEDIA_PLAYER_STOPPED) ) ) {
//设置音频流类型，在 IMediaPlayer.cpp 中对应的 transact 操作是 SET_AUDIO_STREAM_TYPE
        mPlayer->setAudioStreamType(mStreamType);
        //将当前状态设置为 MEDIA_PLAYER_PREPARING
        mCurrentState = MEDIA_PLAYER_PREPARING;
        return mPlayer->prepareAsync();
    }
    ALOGE("prepareAsync called in state %d", mCurrentState);
    return INVALID_OPERATION;
}
```

下面继续分析 `prepareAsync` 函数，`mp->prepareAsync` 对应的 `BnMediaPlayer` 操作如下：

```
MediaPlayerService::Client::prepareAsync
case PREPARE_ASYNC: {
    CHECK_INTERFACE(IMediaPlayer, data, reply);
    reply->writeInt32(prepareAsync());
    return NO_ERROR;
} break;
```

接着分析 `MediaPlayerService::Client::prepareAsync` 函数：

```
status_t MediaPlayerService::Client::prepareAsync()
{
    ALOGV("[%d] prepareAsync", mConnId);
    sp<MediaPlayerBase> p = getPlayer();
```



```
        if (p == 0) return UNKNOWN_ERROR;
        status_t ret = p->prepareAsync();
#ifdef CALLBACK_ANTAGONIZER
        ALOGD("start Antagonizer");
        if (ret == NO_ERROR) mAntagonizer->start();
#endif
        return ret;
    }
```

这里调用了 AwesomePlayer 的 prepareAsync 函数：

```
status_t AwesomePlayer::prepareAsync() {
    ATRACE_CALL();
    Mutex::Autolock autoLock(mLock); // 互斥锁
    if (mFlags & PREPARING) {
        return UNKNOWN_ERROR; // 说明 prepareAsync 已经在等待 prepare 了
    }
    mIsAsyncPrepare = true;
    return prepareAsync_l();
}
```

接着分析 AwesomePlayer::prepareAsync_l 函数：

```
status_t AwesomePlayer::prepareAsync_l() {
    if (mFlags & PREPARING) {
        return UNKNOWN_ERROR; // 说明 prepareAsync 已经在等待 prepare 了
    }
    if (!mQueueStarted) { // 队列不是开始状态时
        mQueue.start(); // 设置成开始状态
        mQueueStarted = true;
    }
    modifyFlags(PREPARING, SET); // 修改状态为 PREPARING
    // 这里 AwesomeEvent 接收到事件，进行回调
    mAsyncPrepareEvent = new AwesomeEvent(
        this, &AwesomePlayer::onPrepareAsyncEvent);
    // 将回调事件通过队列通知出去
    mQueue.postEvent(mAsyncPrepareEvent);
    return OK;
}
```

总结一下上面的代码，首先判断 mFlags，此时不是 PREPARING。接着启动 mQueue（类 TimedEventQueue）。之后修改 mFlags 的状态为 PREPARING，表示现在正在准备处理文件的音视频流。然后实例化一个 AwesomeEvent，放到之前启动的 mQueue 中进行通知。

队列中处理的结果就是调用 AwesomePlayer::onPrepareAsyncEvent 函数。后面的过程就是初始化解码器，将流解码出来，也能知道视频流的宽高属性，然后处于 Prepared 状态，不再



向下跟踪。prepare 的流程就完成了。

接下来，再回到 Java 层中之前的 prepare 函数中的 scanInternalSubtitleTracks 函数：

```
private void scanInternalSubtitleTracks() {
    if (mSubtitleController == null) { //字幕控制器为 null
        Log.d(TAG, "setSubtitleAnchor in MediaPlayer");
        setSubtitleAnchor(); //设置一个字幕控制锚点
    }
    populateInbandTracks();
    if (mSubtitleController != null) {
        mSubtitleController.selectDefaultTrack();
    }
}
```

这个函数用于扫描内嵌字幕并进行跟踪，接下来看看 MediaPlayer 中的 start 函数：

```
public void start() throws IllegalStateException {
    if (isRestricted()) { // 1.判断是否是受限的
        _setVolume(0, 0); // 2.如果受限制，设置声音为 0
    }
    stayAwake(true); // 3.设置唤醒为 1
    _start();
}

private PowerManager.WakeLock mWakeLock = null;
private void stayAwake(boolean awake) {
    if (mWakeLock != null) {
        if (awake && !mWakeLock.isHeld()) {
            mWakeLock.acquire(); //获取
        } else if (!awake && mWakeLock.isHeld()) {
            mWakeLock.release(); //释放
        }
    }
    mStayAwake = awake;
    updateSurfaceScreenOn(); // 4.更新 Surface
}
```

从 Paused 状态变为 Started 状态，如果 playback 已经处于 Stopped 状态，或之前从来没有处于过 Started 状态，playback 将会开始 start。

总结一下上面的代码，start 函数用于 start 或者重新恢复播放，如果 playback 先前已暂停，playback 将从 Paused 状态变为 Started 状态，如果 playback 已经处于 Stopped 状态，或之前从来没有处于过 Started 状态，playback 将会开始 start。

以上的 stayAwake 用于对屏幕进行操作。



首先执行 `PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);`, 通过 `Context.getSystemService` 函数获取 `PowerManager` 实例。然后通过 `PowerManager` 的 `newWakeLock(int flags, String tag)` 来生成 `WakeLock` 实例。`int flags` 指示要获取哪种 `WakeLock`, 不同的锁对 CPU、屏幕、键盘灯有不同的影响。获取 `WakeLock` 实例后通过 `acquire` 获取相应的锁, 然后进行其他业务逻辑的操作, 最后使用 `release` 释放 (释放是必需的)。

关于 `int flags`, 各种锁的类型对 CPU、屏幕、键盘的影响如下。

- `PARTIAL_WAKE_LOCK`: 保持 CPU 运转, 屏幕和键盘灯有可能是关闭的。
- `SCREEN_DIM_WAKE_LOCK`: 保持 CPU 运转, 允许保持屏幕显示但有可能是灰的, 允许关闭键盘灯。
- `SCREEN_BRIGHT_WAKE_LOCK`: 保持 CPU 运转, 允许保持屏幕高亮显示, 允许关闭键盘灯。
- `FULL_WAKE_LOCK`: 保持 CPU 运转, 保持屏幕高亮显示, 键盘灯也保持亮度。
- `ACQUIRE_CAUSES_WAKEUP`: 正常唤醒锁实际上并不打开照明。相反, 一旦打开, 它们会一直保持。当获得 `WakeLock` 时, 这个标志会使屏幕或/和键盘立即打开。一个典型的应用就是, 可以立即看到对用户来说重要的通知。

最后, 通过 `updateSurfaceScreenOn` 函数更新屏幕上的 `Surface`。下面还是回到最上面的 `start` 函数中。在 JNI 中, 对应的是 `android_media_MediaPlayer_start` 函数, 代码如下:

```
static void
android_media_MediaPlayer_start(JNIEnv *env, jobject thiz)
{
    ALOGV("start");
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    //省略空抛异常
    process_media_player_call( env, thiz, mp->start(), NULL, NULL );
}
```

从 `MediaPlayer` 调用 `start` 函数开始, 就进入了视频播放环节, 最终会到 C++ 的 `mediaplayer.cpp` 中实现, 我们先分析一下 `mediaplayer.h` (路径为 `\frameworks\av\include\media\mediaplayer.h`):

```
class MediaPlayer : public BnMediaPlayerClient
{
public:
    MediaPlayer();
    ~MediaPlayer();
    void onFirstRef();
    void disconnect();
```




```
status_t      setDataSource(const char *url);
status_t      setDataSource(int fd, int64_t offset, int64_t length);
status_t      setVideoSurface(const sp<Surface>& surface);
status_t      setListener(const sp<MediaPlayerListener>& listener);
status_t      prepare();
status_t      prepareAsync();
status_t      start();
status_t      stop();
status_t      pause();
bool          isPlaying();
status_t      getVideoWidth(int *w);
status_t      getVideoHeight(int *h);
status_t      seekTo(int msec);
status_t      getCurrentPosition(int *msec);
status_t      getDuration(int *msec);
status_t      reset();
status_t      setAudioStreamType(int type);
status_t      setLooping(int loop);
status_t      setVolume(float leftVolume, float rightVolume);
void          notify(int msg, int ext1, int ext2);
static        sp<IMemory>      decode(const char* url, uint32_t
*pSampleRate, int* pNumChannels);
static        sp<IMemory>      decode(int fd, int64_t offset, int64_t
length, uint32_t *pSampleRate, int* pNumChannels);
//省略部分代码
}
```

从接口中可以看出 MediaPlayer 类实现了一个 MediaPlayer 的基本播放控制操作，如播放（start）、停止（stop）、暂停（pause）、重置（reset）等。

另外的一个类 DeathNotifier 是在 MediaPlayer 类中定义的，它继承了 IBinder 类中的 DeathRecipient 类，这些类都是为进程间通信做准备的：

```
class DeathNotifier: public IBinder:: DeathRecipient
{
public:
    DeathNotifier() {}
    virtual ~DeathNotifier();//析构
    virtual void binderDied(const wp<IBinder>& who);
};
```

对于 MediaPlayerClient 和 MediaPlayerService 通过 IPC 进行通信，后面内容中会进行分析。

可以发现调用 start 函数后，底层返回了一个状态，以便我们知道已经处于 Started 状态还



是没有处于 Started 状态。这时需要用 process_media_player_call 判定这个返回的状态，然后通知 Java 层中的回调事件。

接下来，再分析一下 MediaPlayer 的 pause 函数：

```
public void pause() throws IllegalStateException {
    stayAwake(false); //把唤醒状态置为 false
    _pause(); //调用 native 代码
}
```

在对应的 JNI 中找到 android_media_MediaPlayer_pause 函数：

```
static void
android_media_MediaPlayer_pause(JNIEnv *env, jobject thiz)
{
    ALOGV("pause");
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    //省略判空抛异常
    process_media_player_call( env, thiz, mp->pause(), NULL, NULL );
}
```

查看 pause 函数，可以看到和 start 函数的流程类似，也是通过 mp->pause 返回对应的状态，然后通知上层来暂停的。

2.4 C++中 MediaPlayer 的 C/S 架构

在前面几节中，都是通过 Java 层调用到 JNI 层中，而 JNI 层向下到 C++层并未介绍。本节首先分析 Java 层的一个函数在 C++层 MediaPlayer 中的过程（路径为 frameworks/av/media/libmedia/MediaPlayer.cpp）。

下面找一个我们熟悉的 setDataSource 函数来看看 C（Client）/S（Server）模式的过程。setDataSource 函数如下：

```
status_t MediaPlayer::setDataSource(int fd, int64_t offset, int64_t length)
{
    ALOGV("setDataSource(%d, %" PRId64 ", %" PRId64 ")", fd, offset,
length);
    status_t err = UNKNOWN_ERROR;
    //首先赋值为一个未知错误的状态，就像 boolean 值事先声明为 false 一样
    const sp<IMediaPlayerService>& service(getMediaPlayerService());
    //通过 IMediaPlayerService 获取 Server 端的 MediaPlayerService
    if (service != 0) { //如果 service 不为空
        sp<IMediaPlayer> player(service->create(this, mAudioSessionId));
```




```
//调用 service 的 create 函数
if ((NO_ERROR != doSetRetransmitEndpoint(player)) ||
    (NO_ERROR != player->setDataSource(fd, offset, length))) {
    player.clear();
}
err = attachNewPlayer(player);
}
return err;
}
```

对应看看 MediaPlayerService.cpp 中的 create 函数，MediaPlayerService.cpp 在 C++ 6.0 源码中处于 frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp 中，代码如下：

```
sp<IMediaPlayer> MediaPlayerService::create(const sp<IMediaPlayerClient>&
client, int audioSessionId)
{
    pid_t pid = IPThreadState::self()->getCallingPid();
    int32_t connId = android_atomic_inc(&mNextConnId);
    sp<Client> c = new Client(this, pid, connId, client, audioSessionId,
                             IPThreadState::self()->getCallingUid());
    ALOGV("Create new client(%d) from pid %d, uid %d, ", connId, pid,
          IPThreadState::self()->getCallingUid());
    //表示校验调用方的 uid，用来进行身份验证
    wp<Client> w = c; //把构造的 Client 强引用对象赋值成弱引用对象
    {
        Mutex::Autolock lock(mLock); //互斥锁
        //mClients 声明为 SortedVector< wp<Client> >
        mClients.add(w);
    }
    return c;
}
```

在 new Client 中，有一个 IPThreadState。在 Android 中 ProcessState 是客户端和服务端公共的部分，作为 Binder 通信的基础。ProcessState 是一个 singleton 类，每个进程只有一个对象，这个对象负责打开 Binder 驱动，建立线程池，让其进程里面的所有线程都能通过 Binder 通信。

与之相关的是 IPThreadState，每个线程都有一个 IPThreadState 实例登记在 Linux 线程的上下文附属数据中，主要负责 Binder 的读取、写入和请求处理。IPThreadState 在构造的时候获取进程的 ProcessState 并记录在自己的成员变量 mProcess 中，通过 mProcess 可以获得 Binder 的句柄。IPThreadState 通过 IPThreadState::transact 把 data 及 handle 等填充进 binder_transaction_data，在两个进程间通信。

这里这个 Client 到底是什么？我们又得追踪一下，在 frameworks/av/media/libmediaplayer-service/MediaPlayerService.h 中，如下：

```
class Client : public BnMediaPlayer {
    //IMediaPlayer 接口
    virtual void            disconnect();
    virtual status_t        setVideoSurfaceTexture(const
sp<IGraphicBufferProducer>& bufferProducer);
    virtual status_t        prepareAsync();
    virtual status_t        start();
    virtual status_t        stop();
    virtual status_t        pause();
    virtual status_t        isPlaying(bool* state);
    virtual status_t        setPlaybackSettings(const
AudioPlaybackRate& rate);
    virtual status_t        getPlaybackSettings(AudioPlaybackRate*
rate /* nonnull */);
    virtual status_t        setSyncSettings(const AVSyncSettings&
rate, float videoFpsHint);
    virtual status_t        getSyncSettings(AVSyncSettings* rate /*
nonnull */, float* videoFps /* nonnull */);
    virtual status_t        seekTo(int msec);
    virtual status_t        getCurrentPosition(int* msec);
    virtual status_t        getDuration(int* msec);
    virtual status_t        reset();
    virtual status_t        setAudioStreamType(audio_stream_type_t
type);
    virtual status_t        setLooping(int loop);
    virtual status_t        setVolume(float leftVolume, float
rightVolume);
    //省略部分代码
}; // Client
```

以上代码对应 Java 层的 MediaPlayer 相关方法。如果还记得图 2-3 的话，可以从整体上理解这个 Client 属于什么角色及位置。继承 BnMediaPlayer，并包含了 IMediaPlayer 相关接口。

总结一下上面的代码，Client 类的继承关系为 Client->BnMediaPlayer->IMediaPlayer。分析上面的代码可以看出，create 函数构造了一个 Client 对象，并将此 Client 对象添加到 MediaPlayerService 类的全局列表 mClients 中，这是一个 SortedVector，紧接着执行 player->setDataSource(url, headers)，即 Clients::setDataSource，因此在 setDataSource 中有如下语句：

```
sp<IMediaPlayer> player(service->create(this, mAudioSessionId));
```

等价于


```
sp<IMediaPlayer> player(newClient(**));
```

即 player 最终是用 Client 对象来初始化的，可以直接认为 player==Client。

这时候问题来了，在 C++ 中，这个 Client 及 MediaPlayer 又是什么关系呢？

- Client 是 MediaPlayerService 内部的一个类，我们从上面的代码已知，因为 MediaPlayerService 运行在服务器端，故 Client 也运行在服务器端。
- Client 在 MediaPlayerService.h 中，那接着看看 MediaPlayerService 中的实现，实现过程中调用过 MediaPlayerService 类的一些函数，同样回到 setDataSource。代码如下：

```
status_t MediaPlayerService::Client::setDataSource(
    const sp<IMediaHTTPService> &httpService,
    const char *url,
    const KeyedVector<String8, String8> *headers)
{
    ALOGV("setDataSource(%s)", url);
    if (url == NULL)
        return UNKNOWN_ERROR;
    //这里匹配来自 HTTP、HTTPS、RTSP 的相关 url，这些流是需要经过网络传输的，检查其是否设
    //置了相应的权限，如果没有，返回 PERMISSION_DENIED
    if ((strncmp(url, "http://", 7) == 0) ||
        (strncmp(url, "https://", 8) == 0) ||
        (strncmp(url, "rtsp://", 7) == 0)) {
        if (!checkPermission("android.permission.INTERNET")) {
            return PERMISSION_DENIED;
        }
    }
    //判断是否通过 contentprovider 提供的数据
    if (strncmp(url, "content://", 10) == 0) {
        String16 url16(url);
        int fd = android::openContentProviderFile(url16);
        if (fd < 0)
        {
            ALOGE("Couldn't open fd for %s", url);
            return UNKNOWN_ERROR;
        }
        setDataSource(fd, 0, 0x7fffffffffLL);
        close(fd);
        return mStatus;
    } else {
        player_type playerType = MediaPlayerFactory::getPlayerType(this,
url);
        sp<MediaPlayerBase> p = setDataSource_pre(playerType);
        if (p == NULL) {
```



```

        return NO_INIT;
    }
    setDataSource_post(p, p->setDataSource(httpService, url, headers));
    return mStatus;
}
}

```

接下来重新看看 MediaPlayer 中头文件定义的函数声明，方便对比 Client 中的函数，以下代码在 frameworks/av/include/media/mediaplayer.h 中：

```

class MediaPlayer : public BnMediaPlayerClient, public virtual
IMediaDeathNotifier
{
public:
    MediaPlayer();
    ~MediaPlayer();

    void died();
    void disconnect();
    status_t setDataSource(
        const sp<IMediaHTTPService> &httpService,
        const char *url,
        const KeyedVector<String8, String8> *headers);
    status_t setDataSource(int fd, int64_t offset, int64_t
length);
    status_t setDataSource(const sp<IStreamSource>
&source);
    status_t setDataSource(const sp<IDataSource> &source);
    status_t setVideoSurfaceTexture(const sp<IGraphic-
BufferProducer> &bufferProducer);
    status_t setListener(const sp<MediaPlayerListener>&
listener);

    status_t prepare();
    status_t prepareAsync();
    status_t start();
    status_t stop();
    status_t pause();
    bool isPlaying();
    //省略部分代码
};

```

这里的函数和 Client 中的函数是一一对应的，两者通过 Client 的代理类联系在了一起：

```

status_t MediaPlayer::setDataSource(
    const sp<IMediaHTTPService> &httpService,
    const char *url, const KeyedVector<String8, String8> *headers)
{

```

```

        ALOGV("setDataSource(%s)", url);
        status_t err = BAD_VALUE;
        if (url != NULL) {
            const sp<IMediaPlayerService>& service(getMediaPlayerService());
            if (service != 0) {
                sp<IMediaPlayer> player(service->create(this, mAudioSessionId));
                if ((NO_ERROR != doSetRetransmitEndpoint(player)) ||
                    (NO_ERROR != player->setDataSource(httpService, url,
headers))) {
                    player.clear();
                }
                err = attachNewPlayer(player);
            }
        }
        return err;
    }

status_t MediaPlayer::attachNewPlayer(const sp<IMediaPlayer>& player)
{
    status_t err = UNKNOWN_ERROR;
    //这里就是 Client 声明的在客户端处的代理类
    sp<IMediaPlayer> p;
    {
        Mutex::Autolock _l(mLock); //加锁
        if ( !( (mCurrentState & MEDIA_PLAYER_IDLE) ||
                (mCurrentState == MEDIA_PLAYER_STATE_ERROR) ) ) {
            ALOGE("attachNewPlayer called in state %d", mCurrentState);
            return INVALID_OPERATION;
        }
        clear_l();
        p = mPlayer;
//赋值给代理类, mPlayer 在 MediaPlayer.h 中声明, 其是一个 Client 在客户端处的代理类对象
        mPlayer = player;
        if (player != 0) {
            mCurrentState = MEDIA_PLAYER_INITIALIZED;
            err = NO_ERROR;
        } else {
            ALOGE("Unable to create media player");
        }
    }

    if (p != 0) {
        p->disconnect();
    }
}

```



```

    return err;
}

```

上面的两个函数，一个是 MediaPlayer 的 setDataSource，会调到 attachNewPlayer 函数，这个函数最终会调用服务器端 Client 对应的函数。到这里可能有读者会想，IMediaPlayer.h 和 mediaplayer.h 的区别是什么？那么下面介绍一下 IMediaPlayer.h、mediaplayer.h、IMediaPlayerClient.h 的区别。

- 从包结构上看：IMediaPlayer 和 IMediaPlayerClient.h 都在 frameworks/av/media/libmedia 包中，而 mediaplayer.h 在 /av/include/media 包中（前面已有代码贴出）。
- 从功能上看：它们肩负的职责也不一样。

这里贴出 IMediaPlayer.h 及 IMediaPlayerClient.h 的代码，IMediaPlayer.h 位于 frameworks/av/media/libmedia 包中：

```

#ifndef ANDROID_IMEDIAPLAYER_H
#define ANDROID_IMEDIAPLAYER_H

#include <utils/RefBase.h>
#include <binder/IInterface.h>
#include <binder/Parcel.h>
#include <utils/KeyedVector.h>
#include <system/audio.h>
struct sockaddr_in;
namespace android {
class IMediaPlayer: public IInterface
{
public:
    DECLARE_META_INTERFACE(MediaPlayer);
    virtual void disconnect() = 0;
    virtual status_t setDataSource(const sp<IDataSource>& source)
= 0;
    virtual status_t setVideoSurfaceTexture(const sp<IGraphic-
BufferProducer>& bufferProducer) = 0;
    virtual status_t prepareAsync() = 0;
    virtual status_t start() = 0;
    virtual status_t stop() = 0;
    virtual status_t pause() = 0;
    //省略部分代码
};

class BnMediaPlayer: public BnInterface<IMediaPlayer>
{
public:

```



```

        virtual status_t    onTransact( uint32_t code,
                                        const Parcel& data,
                                        Parcel* reply,
                                        uint32_t flags = 0);
    };

}; //namespace android
#endif // ANDROID_IMEDIAPLAYER_H

```

在 IMediaPlayer.h 中定义的基本上都是虚函数，而我们知道虚函数在 C++ 中用于实现多态性 (Polymorphism)，多态性是将接口与具体实现代码进行了分离，用形象的语言来解释就是以共同的方法实现，但因个体差异而采用不同的策略。所以它的功能是实现 MediaPlayer 功能的接口，看到 onTransact 函数，自然联想 Binder 通信，把底层的 Parcel 指针类型数据向上层的另一个进程传递。

再分析一下 IMediaPlayerClient.h，同样位于 frameworks/av/media/libmedia 包中：

```

#include <utils/RefBase.h>
#include <binder/IInterface.h>
#include <binder/Parcel.h>
#include <media/IMediaPlayerClient.h>
namespace android {
enum {
    NOTIFY = IBinder::FIRST_CALL_TRANSACTION,
};

class BpMediaPlayerClient: public BpInterface<IMediaPlayerClient>
{
public:
    BpMediaPlayerClient(const sp<IBinder>& impl)
        : BpInterface<IMediaPlayerClient>(impl)
    {
    }

    virtual void notify(int msg, int ext1, int ext2, const Parcel *obj)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IMediaPlayerClient::
getInterfaceDescriptor());
        data.writeInt32(msg);
        data.writeInt32(ext1);
        data.writeInt32(ext2);
        if (obj && obj->dataSize() > 0) {
            data.appendFrom(const_cast<Parcel *>(obj), 0, obj->dataSize());

```

```

    }
    remote()->transact(NOTIFY, data, &reply, IBinder::FLAG_ONEWAY);
}
};

IMPLEMENT_META_INTERFACE(MediaPlayerClient, "android.media.
IMediaPlayerClient");
status_t BnMediaPlayerClient::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch (code) {
        case NOTIFY: {
            CHECK_INTERFACE(IMediaPlayerClient, data, reply);
            int msg = data.readInt32();
            int ext1 = data.readInt32();
            int ext2 = data.readInt32();
            Parcel obj;
            if (data.dataAvail() > 0) {
                obj.appendFrom(const_cast<Parcel*>(&data), data.
dataPosition(), data.dataAvail());
            }
            notify(msg, ext1, ext2, &obj);
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
} //namespace android

```

总结一下上面的代码，在内部定义一个 BpMediaPlayerClient 类（也就是 Client 的父类），然后它有一个 onTransact 函数。一般 onXXX 都是被动回调过来的，不是由自己控制的，如 Activity 中的 onCreate、onPause、onStart 函数，这些函数都是其他地方处理并通知到 Activity 中的。这里也是一样的，onTransact 作为 Binder 通信中的回调函数，前面介绍到 player 实际上是 C/S 模式的，IMediaPlayerClient.h 的功能是描述一个 MediaPlayer 客户端的接口。

综上所述，mediaplayer.h 的功能是对外（JNI 层）的接口类，它最主要的是定义了一个 MediaPlayer 类（C++层），我们在 android_media_MediaPlayer.cpp 中就引入了 media/mediaplayer.h；IMediaPlayer.h 则是一个实现 MediaPlayer（C++层）功能的接口；而 IMediaPlayerClient.h 的功能是描述一个 MediaPlayer 客户端（这里暂且理解为前面说的 Client）的接口。

第 3 章

管理调度的服务者

MediaPlayerService

MediaPlayerService 是多媒体框架中一个非常重要的服务，从前面的内容中我们可以理解 MediaPlayer 是客户端，MediaPlayerService 和 MediaPlayerService::Client 是服务器端。MediaPlayerService 实现 IMediaPlayerService 定义的业务逻辑，其主要功能是根据 MediaPlayer::setDataSource 输入的 URL 调用 create 函数创建对应的 player。MediaPlayerService::Client 实现 IMediaPlayer 定义的业务逻辑，其主要功能包括 start、stop、pause、resume……其是通过调用 MediaPlayerService create 的 player 中的对应方法来实现具体功能的。

3.1 Client/Server 通过 IPC 的通信流程图

我们知道一般 Android 中很多通信是通过 Binder 机制来完成的，对于 MediaPlayer 和 MediaPlayerService 来说也不例外。我们通过图 3-1 来看看它们是如何通信的。

从图 3-1 中可以总结出如下几点。

- MediaPlayer 是客户端，也就是我们所说的 C/S 模型中的 C 端，即 Client。
- MediaPlayerService 和 MediaPlayerService::Client 是服务器端，也就是我们所说的 C/S 模型中的 S 端，即 Server 端。
- MediaPlayerService 实现 IMediaPlayerService 定义的业务逻辑，其主要功能是根据 MediaPlayer::setDataSource 输入的 URL 调用 create 函数创建对应的 player。

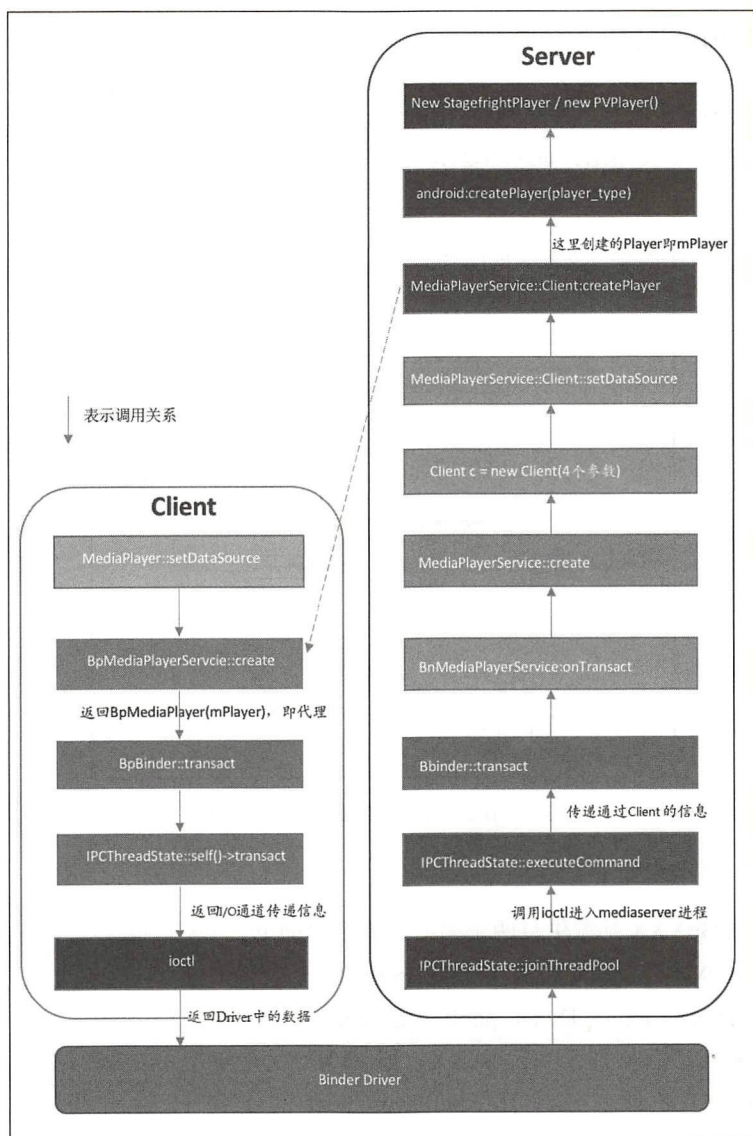


图 3-1 MediaPlayer 和 MediaPlayerService 通过 Binder 通信

- MediaPlayerService::Client 实现 IMediaPlayer 定义的业务逻辑，其主要功能包括 start、stop、pause、resume……其是通过调用 MediaPlayerService create 的 player 中的对应方法来实现具体功能的。
- 通过 Transact 函数可以向远端的 IBinder 对象发出调用，通过 onTransact 函数可以使你自己的远程对象能够响应接收到的调用。

3.2 相关联的类图

我们知道 Binder 通信时，需要通过 IBinder 接口转化具体的实体对象，那么中间会有很多的类，下面看看类关系图，如图 3-2 所示。

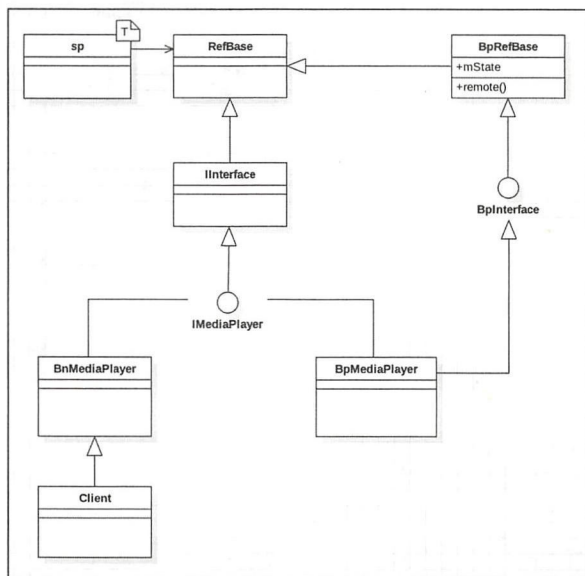


图 3-2 IPC 通信相关联的类关系 (1)

从图 3-2 可以总结为如下几点。

- BnXXX 或 BpXXX 都派生自两个类，具体情况如下。
 - `class BpXXX : public IXXX, public BpRefBase`
 - `class BnXXX : public IXXX, public BBinder`
- BpXXX 和 BnXXX 都派生自 IXXX，那 IXXX 又是做什么的呢？这里可以理解为定义业务逻辑，我们此前分析 IMediaPlayerClient 的作用时，也介绍过。但 BpXXX 与 BnXXX 中的实现方式不同。
 - 在 BpXXX 中，把对应的 `binder_transaction_data` 打包之后，通过 BpRefBase 中的 `mRemote(BpBinder)` 发送出去，并等待结果。
 - 在 BnXXX 中，实现对应的业务逻辑，通过调用 BnXXX 派生类中的方法来实现，如 `MediaPlayerService::Client`。
- 从图 3-3 可以看出，IBinder 用于进行进程间通信。
 - 图 3-2 中的 BpRefBase 中有一个 `remote` 函数用来与 Binder 驱动交互使用。

- Binder 是用来从 Binder 驱动中接收相关请求并进行相关处理的。
- BpBinder 和 BinderDriver 进行互通。

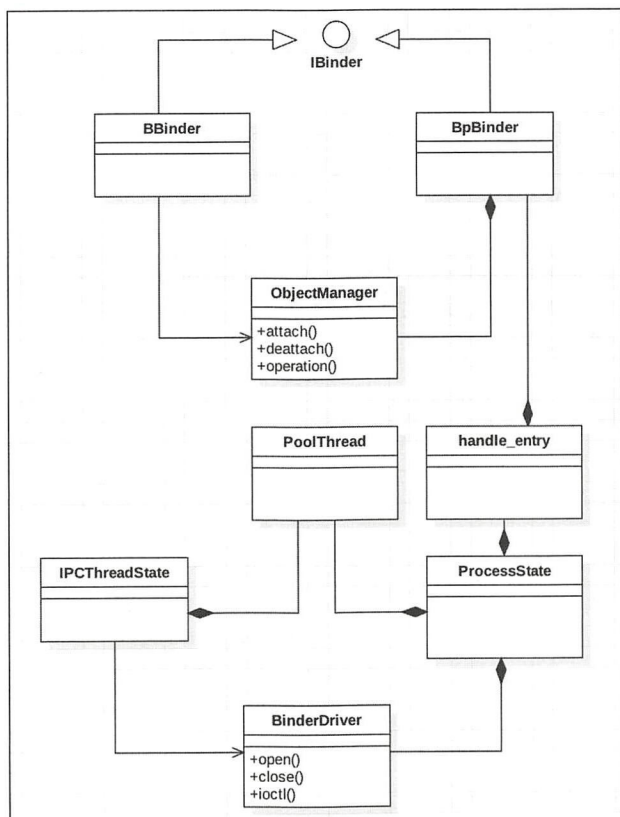


图 3-3 IPC 通信相关联的类关系 (2)

3.3 产生过程

在了解 MediaPlayerService 之前，先了解一下 IMediaPlayerService.cpp，在 C++ 6.0 源码中其处于 frameworks/av/media/libmediaplayerservice/MediaPlayerService.cpp 中：

```

#ifndef ANDROID_IMEDIAPLAYERSERVICE_H
#define ANDROID_IMEDIAPLAYERSERVICE_H

namespace android {
class IMediaPlayerService: public IInterface
{

```


Android 音视频开发

```

public:
    DECLARE_META_INTERFACE(MediaPlayerService);
    virtual sp<IMediaRecorder> createMediaRecorder(const String16
&opPackageName) = 0;
    virtual sp<IMediaMetadataRetriever> createMetadataRetriever() = 0;
    virtual sp<IMediaPlayer> create(const sp<IMediaPlayerClient>& client,
int audioSessionId = 0) = 0;
    virtual sp<IOMX> getOMX() = 0;
    virtual sp<ICrypto> makeCrypto() = 0;
    virtual sp<IDrm> makeDrm() = 0;
    virtual sp<IHDCP> makeHDCP(bool createEncryptionModule) = 0;
    virtual sp<IMediaCodecList> getCodecList() const = 0;
    virtual sp<IRemoteDisplay> listenForRemoteDisplay(const String16
&opPackageName, const sp<IRemoteDisplayClient>& client, const String8& iface)
= 0;

    //省略部分代码
};
// -----
class BnMediaPlayerService: public BnInterface<IMediaPlayerService>
{
public:
    virtual status_t    onTransact( uint32_t code,
                                   const Parcel& data,
                                   Parcel* reply,
                                   uint32_t flags = 0);
};
}; // namespace android
#endif // ANDROID_IMEDIAPLAYERSERVICE_H

```

可以看出这里定义了一些常规播放控制接口，接下来开始了解 MediaPlayerService，首先找到入口，在 frameworks/base/media/mediaserver/main_mediaserver.cpp 中：

```

int main(int argc __unused, char** argv)
{
    //省略部分代码
    if (doLog && (childPid = fork()) != 0) {
        //省略部分代码
        sp<ProcessState> proc(ProcessState::self());
        //获得 ProcessState，在构造函数中打开 Binder 驱动
        MediaLogService::instantiate();
        ProcessState::self()->startThreadPool();
        for (;;) {
            //省略部分代码
            sp<IServiceManager> sm = defaultServiceManager();

```

第3章 管理调度的服务者 MediaPlayerService

```

        sp<IBinder> binder = sm->getService(String16("media.log"));
        if (binder != 0) {
            Vector<String16> args;
            binder->dump(-1, args);
        }
        //省略部分代码
    }
} else {
    //其他的 Service 实例化
    InitializeIcuOrDie();
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    ALOGI("ServiceManager: %p", sm.get());
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
}

```

接着看看 defaultServiceManager 函数，代码如下：

```

sp<IServiceManager> defaultServiceManager()
{
    if(gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
        AutoMutex -l(gDefaultServiceManagerLock);
        if(gDefaultServiceManager == NULL) //获取 ServiceManager
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
    }
    return gDefaultServiceManager;
}

```

用的是一个单例，每个进程只需要一个 BpServiceManager 代理，ProcessState::self() -> getContextObject(NULL)，接下来看看 getContextObject(NULL)函数，看看 ProcessState::self() -> getContextObject(NULL)，代码如下：

```

sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{
    return getStrongProxyForHandle(0);
}
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;

```


Android 音视频开发

```

    AutoMutex _l(mLock);
    //在数组 mHandleToObject 里面根据 handle 索引查找 handle_entry 结构体
    handle_entry *e = lookupHandleLocked(handle);//1
    if( e != NULL) {
        IBinder* b = e->binder;// 2.指向结构体中的 binder 地址
        if(b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);// 3.相当于 BpBinder(0)
            e->binder = b;
            if(b) e->refs = b->getWeakRefs();
            result = b;
        }else{
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result;
}
struct handle_entry{
    IBinder* binder;
    RefBase::weakref_type* refs;
}

```

总结一下上面的代码，传入的句柄 handle 值为 0，表示 ServiceManager，构建一个 BpBinder，所以现在相当于 `gDefaultServiceManager = interface_cast(new BpBinder(0));`。

接下来看看 `interface_cast` 是什么，其处于 `frameworks/base/include/binder/IInterface.h` 中：

```

template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}

```

等价于：

```

inline sp<IServiceManager> interface_cast(const sp<IBinder>& obj)
{
    return IServiceManager::asInterface(obj);
}

```

下面继续跟到 `IServiceManager` 里面去，其处于 `frameworks/base/include/binder/IServiceManager.h` 中：

```

class IServiceManager:public IInterface
{
public:

```


第3章 管理调度的服务者 MediaPlayerService

```

DECLARE_META_INTERFACE(ServiceManager); // 宏
virtual status_t addService(const String16& name, const sp<IBinder>&
service) = 0;
virtual sp<IBinder> getService(const String16& name) const = 0;
}
//IInterface 是可以被多个类继承的
{
    const android::String16 IServiceManager::descriptor(NAME);
    const android::String16
        IServiceManager::getInterfaceDescriptor() const {
        return IServiceManager::descriptor;
    }
    android::sp<IServiceManager> IServiceManager::asInterface(
        const android::sp<android::IBinder>& obj)
        //参数为 new BpBinder(0)
    {
        android::sp<IServiceManager> intr;
        if (obj != NULL) {
            intr = static_cast<IServiceManager*>(
                obj->queryLocalInterface(
                    IServiceManager::descriptor).get());
            if (intr == NULL) {
                intr = new BpServiceManager(obj);
                //这里构造了一个 BpServiceManager 对象
            }
        }
        return intr;
    }
    IServiceManager::IServiceManager() { }
    IServiceManager::~IServiceManager() { }
}

```

总结一下，句柄 `handle(0)` 创建了一个 `new BpBinder(0)`，根据这个 `BpBinder` 创建一个 `BpServiceManager` 代理。

下面来分析 `BpServiceManager`，代码如下：

```

class BpServiceManager : public BpInterface<IServiceManager>
{
public:
    BpServiceManager(const sp<IBinder>& impl) : BpInterface<
IServiceManager>(impl)
    {}
}

```

Android 音视频开发

这里的 BpInterface 是一个模板类，表示 BpServiceManager 同时继承于 BpInterface 和 IServiceManager 类：

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public: BpInterface(const sp<IBinder>& remote);
...
}
```

调用了基类 BpInterface 的构造函数：

```
BpInterface<IServiceManager>::BpInterface(const sp<IBinder>& remote) :
BpRefBase(remote) {}
//这里的 remote 就是构造的 BpBinder
BpRefBase::BpRefBase(const sp<IBinder>& o) : mRemote(o.get()), mRefs
(NULL), mState(0) {}
```

开始添加服务内部，在 frameworks/base/media/libmediaplayerservice/MediaPlayerService.cpp 中，有一个 instantiate 函数，通过 ServiceManager 来添加 MediaPlayerService 服务：

```
void MediaPlayerService::instantiate()
{
    defaultServiceManager()->addService(String16("media.player"), new
MediaPlayerService);
}
```

defaultServiceManager 返回的是刚创建的 BpServiceManager，并调用 add 函数。

BpMediaPlayerService 作为服务代理端，那么 BnMediaPlayerService 一定是实现端，MediaPlayerService 继承自 BnMediaPlayerService，实现了真正的业务函数。

3.4 添加服务的过程

下面分析一下 BpServiceManager 的 addService 函数：

```
virtual status_t addService(const String16& name, const sp<IBinder>&
service)
{
    Parcel data, reply;
    data.writeInterfaceToken(IServiceManager.getInterfaceDescriptor());
    //android.os.IServiceManager
    data.writeString16(name); //media.player
    data.writeStrongBinder(service); //也就是 MediaPlayerService
```


第3章 管理调度的服务者 MediaPlayerService

```

        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data,
&reply);
        return err == NO_ERROR ? reply.readInt32() : err;
    }

```

这里的 `remote` 函数返回的就是前面创建的 `BpBinder(0)` 对象：

```

status_t BpBinder::transact(uint32_t code, const Parcel& data, Parcel*
reply, uint32_t flags)
{
    IPCThreadState::self()->transact(mHandle, code, data, reply, flags);
}

status_t IPCThreadState::transact(int32_t handle, uint32_t code, const
Parcel& data, Parcel* reply, uint32_t flags)
{
    //发送 ADD_SERVICE_TRANSACTION 请求
    writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    if(reply) //等待响应
        waitForResponse(NULL, reply);
}

status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t
binderFlags, int32_t handle, uint32_t code, const Parcel& data, status_t *
statusBuffer)
{
    //cmd    BC_TRANSACTION    应用程序向 Binder 发送的命令
    binder_transaction_data tr;
    tr.target.handle = handle; //0
    tr.code = code;           //ADD_SERVICE_TRANSACTION
    tr.flags = binderFlags;
    //把命令和数据一起发送到 Parcel mOut 中
    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));
}

status_t IPCThreadState::waitForResponse(Parcel* reply, status_t
*acquireResult)
{
    int32_t cmd;
    while(1)
        talkWithDriver();
    cmd = mIn.readInt32();
    switch(cmd) {
        case BR_TRANSACTION_COMPLETE:
            ...
            break;
    }
}

```


Android 音视频开发

```

    }
    {
        return err;
    }

```

接着看看 `talkWithDriver` 的实现，顾名思义，是与 `driver` 谈话：

```

status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    binder_write_read bwr;
    bwr.write_size = outAvail;
    bwr.write_buf = (long unsigned int)mOut.data();    //写入 mOut 的数据
    bwr.read_size = mIn.dataCapacity;
    bwr.read_buffer = (long unsigned int)mIn.data();
    ioctl(mProcess->mDriverFD, BINDER_WRITE_READm &bwr);
    //把 mOut 写到 Binder，并读取 mIn 的数据
}

```

`IPCThreadState::joinThreadPool`、`ProcessState::self->startThreadPool` 进入线程循环，`talkWithDriver` 等待客户端 `Client` 请求，从 `Binder` 读取命令请求进行处理。

到现在为止，`MediaPlayerService` 的服务器端已经向服务总管 `ServiceManager` 注册了。

3.5 通过 `BinderDriver` 和 `MediaPlayer` 通信的过程

下面看看客户端是如何获得服务的代理并和服务器端通信的。我们以 `MediaPlayer` 的业务函数解码解析播放一个网络视频的 URL 为例：

```

sp<IMemory> MediaPlayer::decode(const char*url, uint32_t *pSampleRate, ...)
{
    sp<IMemory> p;
    const sp<IMediaPlayerService>& service = getMediaPlayerService();
    //获得 BpMediaPlayerService 代理
    if(service != 0)
        p = service->decode(url, ....);
    return p;
}

```

这里主要分析 `getMediaPlayerService`，即客户端是如何向 `ServiceManager` 总管查询服务并获得代理的：

```

sp<IMediaPlayerService>& IMediaDeathNotifier::getMediaPlayerService()
{
    sp<IServiceManager> sm = defaultServiceManager();
}

```

第3章 管理调度的服务者 MediaPlayerService

```

//生成一个 BpServiceManager 代理对象
sp<IBinder> binder;
do {
    //获取 String16("media.player") 的服务对象
    binder = sm->getService(String16("media.player"));
    if(binder != 0)
        break;
    usleep(500000)
} while(true);
sMediaPlayerService = interface_cast<IMediaPlayerService>(binder);
return sMediaPlayerService;
}

```

先获得 BpServiceManager 的代理，然后调用 getService 函数向服务总管 ServiceManager 查询名叫 String16("media.player")的服务，其处于 frameworks/base/libs/binder/IServiceManager.cpp 中：

```

class BpServiceManager : public BpInterface<IServiceManager>
{
public:
    virtual sp<IBinder> getService(const String16& name) const
    {
        for(n = 0; n < 5; n++) {
            sp<IBinder> svc = checkService(name); //调用 checkService 函数
            if(svc != NULL) return svc;
            sleep(1);
        }
        return NULL;
    }
    virtual sp<IBinder> checkService(const String16& name) const
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterface
Descriptor());
        //首先调用 data.writeInt32(IPCThreadState::self()->getStrictModePolicy())
        //然后写入 android.os.IServiceManager
        data.writeString16(name); //写入 media.player
        remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
        //然后通过 Binder 把 data 及 reply 的数据通知出去，在 onTransact 函数中进行回调
        return reply.readStrongBinder();
    }
}

```

这里首先将请求打包成 Parcel 格式，然后调用 remote->transact 函数，前面我们分析过

BpServiceManager::remote 返回的就是 new BpBinder(0)，对应的句柄为 ServiceManager。继续在 BpBinder 中寻找实现代码，其处于 frameworks/base/libs/binder/BpBinder.cpp 中：

```
status_t BpBinder::transact(uint32_t code, const Parcel& data, Parcel* reply,
uint32_t flags)
{
    IPCThreadState::self()->transact(mHandle, code, data, reply, flags);
}
```

最后调用 IPCThreadState 的 transact 函数，在 Android 中 ProcessState 是客户端和服务端公共的部分，作为 Binder 通信的基础，ProcessState 是一个 singleton 类，每个进程只有一个对象，这个对象负责打开 Binder 驱动，建立线程池，让其进程里面的所有线程都能通过 Binder 通信。

与之相关的是 IPCThreadState，每个线程都有一个 IPCThreadState 实例登记在 Linux 线程的上下文附属数据中，主要负责 Binder 的读取、写入和请求处理框架。IPCThreadState 在构造的时候获取进程的 ProcessState 并记录在自己的成员变量中：

```
status_t IPCThreadState::transact(int32_t handle, uint32_t code, const
Parcel& data, Parcel* reply, uint32_t flags)
{
    //填充 binder_transaction_data 结构体，写入 mOut 中
    writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    //调用 talkWithDriver()，将 mOut 写入 Binder 驱动，并从 Binder 驱动读取 mIn 数据
    waitForResponse(reply);
}
```

通过 writeTransactionData 函数来填充 mOut 结构体，mOut 里面的内容如下：

```
mOut.writeInt32(BC_TRANSACTION);
mOut.write(&tr, sizeof(tr));
```

这里 binder_transaction_data tr 的内容如下：

```
tr.target.handle = 0; //表面是发往 ServiceManager 的
tr.code = CHECK_SERVICE_TRANSACTION;
tr.flags = 0;
```

tr.data 的内容如下：

```
data.writeInt32(IPCThreadState::self()->getStrictModePolicy() | STRICT_
MODE_PENALTY_GATHER);
data.writeString16("android.os.IServiceManager");
data.writeString16("media.player");
```

waitForResponse 函数是关键：


```

status_t IPCThreadState::waitForResponse(Parcel* reply)
{
    while(1) {
        talkWithDriver();    //输入 mOut, 输出 mIn
        cmd = mIn.readInt32();
        switch(cmd) {
            case BR_REPLY:
            {
                binder_transaction_data tr;
                mIn.read(&tr, sizeof(tr));
                if(reply) {
                    reply->ipcSetDataReference(reinterpret_cast<const uint8_t*>
(tr.data.ptr.buffer), tr.data.size, reinterpret_cast<const size_t*>
(tr.data.ptr.offsets), tr.offsets_size/sizeof(size_t), freeBuffer, this);
                } else {
                    err = *static_cast<const status_t*>(tr.data.ptr.buffer);
                    freeBuffer(NULL, reinterpret_cast<const uint8_t*>(tr.data.
ptr.buffer), tr.data.size, reinterpret_cast<const size_t*>(tr.data.ptr.
offsets), tr.offsets_size/sizeof(size_t), freeBuffer, this)
                }
            }
        }
    }
}

```

返回的是 `return reply.readStrongBinder();`, 进入 Parcel 的 `readStrongBinder` 函数:

```

sp<IBinder> Parcel::readStrongBinder() const
{
    sp<IBinder> val;
    unflatten_binder(ProcessState::self(), *this, &val);
    return val;
}

status_t unflatten_binder(const sp<ProcessState>& proc, const Parcel& in,
sp<IBinder>* out)
{
    const flat_binder_object* flat = in.readObject(false);
    if(flat) {
        switch(flat->type) {
            case BINDER_TYPE_BINDER:
                *out = static_cast<IBinder*>(flat->cookie);
                return finish_unflatten_binder(NULL, *flat, in);
            case BINDER_TYPE_HANDLE:
                *out = proc->getStrongProxyForHandle(flat->handle);

```

```

        return finish_unflatten_binder(static_cast<BpBinder*> (out->
get()), *flat, in);
    }
}
}

```

这里的 flat->type 是 BINDER_TYPE_HANDLE，所以调用 ProcessState::getStrongProxyForHandle 函数：

```

sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    handle_entry* e = lookupHandleLocked(handle);
    if(e != NULL) {
        IBinder* b = e->binder;
        if(b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);
            e->binder = b;
            if( b ) e->refs = e->getWeakRefs();
            result = b;
        } else {
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result;
}

```

这里的 handle 就是 ServiceManager 内维护的 MediaPlayerService 对应的 Binder 句柄，这个 ProcessState 根据这个句柄新建了一个 BpBinder，并将其保存起来，这样下次需要从 ServiceManager 请求获取相同句柄的时候就可以直接返回了。

根据这个返回的 BpBinder 获得 MediaPlayerService 的代理：

```

sMediaPlayerService = interface_cast(binder);

```

和前面的 ServiceManager 一样，调用 IMediaPlayerService 的 asInterface 宏函数：

```

android::sp<IMediaPlayerService> IMediaPlayerService::asInterface(const
android::sp<android::IBinder>& obj)
{
    android::sp<IMediaPlayerService> intr;
    if(obj != NULL) {
        intr = static_cast<IMediaPlayerService>(
            obj->queryLocalInterface(IMediaPlayerService::descriptor).get);
        if (intr == NULL) {
            intr = new BpMediaPlayerService(obj);
        }
    }
}

```



```

    }
    return intr;
}

```

这样就获得了一个代理 BpMediaPlayerService 对象，它的 remote 为 BpBinder(handle)，这个 handle 就是向服务总管 ServiceManager 查询到的 MediaPlayerService 对应的 Binder 句柄。

最后总结一下：

- 在实际业务中，如当 MediaPlayer::setDataSource 返回时，会创建一个与 MediaPlayerService::Client 对应的 BpMediaPlayer，用于获取 MediaPlayerService::Client 的各项功能。
- MediaPlayer 又是如何找到 MediaPlayerService::Client 的呢？只有 MediaPlayerService 向 ServiceManager 进行了注册才可以，所以 MediaPlayer 必须先获取 BpMediaPlayerService，然后通过 BpMediaPlayerService 的管理功能创建一个 MediaPlayerService::Client。
- 为什么不直接定义一个 MediaPlayer，让其向 ServiceManager 注册呢？MediaPlayerService 包含的功能不仅是 Client，还有 AudioOutput、AudioCache、MediaConfigClient 功能。MediaPlayerService 就是一个媒体服务的窗口（Driver 有点类似一个场地，在这个场地内沟通好信息），MediaPlayerService 把生意谈好，合同签回来，再根据合同上的要求安排不同的开发人员去做。

3.6 创建播放器

在 MediaPlayer 的构造函数中，会预先注册一些播放器的 Factory。很多 ROM 厂商如果要应用自己写的播放器，也需要有一个 Factory，如小米盒子 ROM 引入了 VLC 多媒体框架，这里也是一样的：

```

MediaPlayerService::MediaPlayerService()
{
    ALOGV("MediaPlayerService created");
    mNextConnId = 1;
    mBatteryAudio.refCount = 0;
    for (int i = 0; i < NUM_AUDIO_DEVICES; i++) {
        mBatteryAudio.deviceOn[i] = 0;
        mBatteryAudio.lastTime[i] = 0;
        mBatteryAudio.totalTime[i] = 0;
    }
    mBatteryAudio.deviceOn[SPEAKER] = 1;
    BatteryNotifier& notifier(BatteryNotifier::getInstance());
    notifier.noteResetVideo();
}

```



```

    notifier.noteResetAudio();
    MediaPlayerFactory::registerBuiltinFactories();
    //注册一些播放器的 Factory
}

```

看看 registerBuiltinFactories 函数，会注册两个 Factory：

```

void MediaPlayerFactory::registerBuiltinFactories() {
    Mutex::Autolock lock_(&sLock);
    if (sInitComplete)
        return;
    registerFactory_1(new StagefrightPlayerFactory(), STAGEFRIGHT_ PLAYER);
    registerFactory_1(new NuPlayerFactory(), NU_ PLAYER);
    registerFactory_1(new TestPlayerFactory(), TEST_ PLAYER);
    sInitComplete = true;
}

```

通过 StagefrightPlayerFactory 创建一个 StagefrightPlayer，代码如下：

```

class StagefrightPlayerFactory : public MediaPlayerFactory::IFactory {
public:
    virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/,int fd,
int64_t offset, int64_t length, float /*curScore*/) {
        if (legacyDrm()) {
            sp<DataSource> source = new FileSource(dup(fd), offset, length);
            String8 mimeType;
            float confidence;
            if (SniffWVM(source, &mimeType, &confidence, NULL /* format
*/) {
                return 1.0;
            }
        }

        if (getDefaultPlayerType() == STAGEFRIGHT_ PLAYER) {
            char buf[20];
            lseek(fd, offset, SEEK_SET);
            read(fd, buf, sizeof(buf));
            lseek(fd, offset, SEEK_SET);
            uint32_t ident = *((uint32_t*)buf);
            //表示 OGG Vorbis, OGG Vorbis 是一种音频压缩格式
            if (ident == 0x5367674f) //'OggS'
                return 1.0;
        }
        return 0.0;
    }
    virtual sp< > createPlayer(pid_t /* pid */) {

```

```

        ALOGV(" create StagefrightPlayer");
        return new StagefrightPlayer();
    }
    //省略部分代码
};

```

通过 NuPlayerFactory 可以创建一个 NuPlayer，代码如下：

```

class NuPlayerFactory : public MediaPlayerFactory::IFactory {
public:
    virtual float scoreFactory(const sp<IMediaPlayer>& /*client*/, const
char* url, float curScore) {
        static const float kOurScore = 0.8;
        if (kOurScore <= curScore)
            return 0.0;
        //是 http://, 或是 https://, 或是 file://
        if (!strncasecmp("http://", url, 7)
            || !strncasecmp("https://", url, 8)
            || !strncasecmp("file://", url, 7)) {
            size_t len = strlen(url);
            //后缀是 m3u8
            if (len >= 5 && !strcasecmp(".m3u8", &url[len - 5])) {
                return kOurScore;
            }
            if (strstr(url, "m3u8")) {
                return kOurScore;
            }
            //如果是 SDP 协议
            if ((len >= 4 && !strcasecmp(".sdp", &url[len - 4])) ||
                strstr(url, ".sdp?")) {
                return kOurScore;
            }
        }
        //如果是 RTSP 协议
        if (!strncasecmp("rtsp://", url, 7)) {
            return kOurScore;
        }
        return 0.0;
    }
    //省略部分代码
    virtual sp<MediaPlayerBase> createPlayer(pid_t pid) {
        ALOGV(" create NuPlayer");
        return new NuPlayerDriver(pid);
    }
};

```

这里并没有直接创建一个 NuPlayer，而是创建了 NuPlayerDriver，实际上是在 NuPlayerDriver 的构造函数中创建了一个 NuPlayer。

3.7 建立 StageFright 层交互

前面章节中建立的 NuPlayer 需要和 StageFright 层进行交互，那么 NuPlayer 以及 StageFright 层的关系是怎样的呢？下面通过图 3-4 来看看。

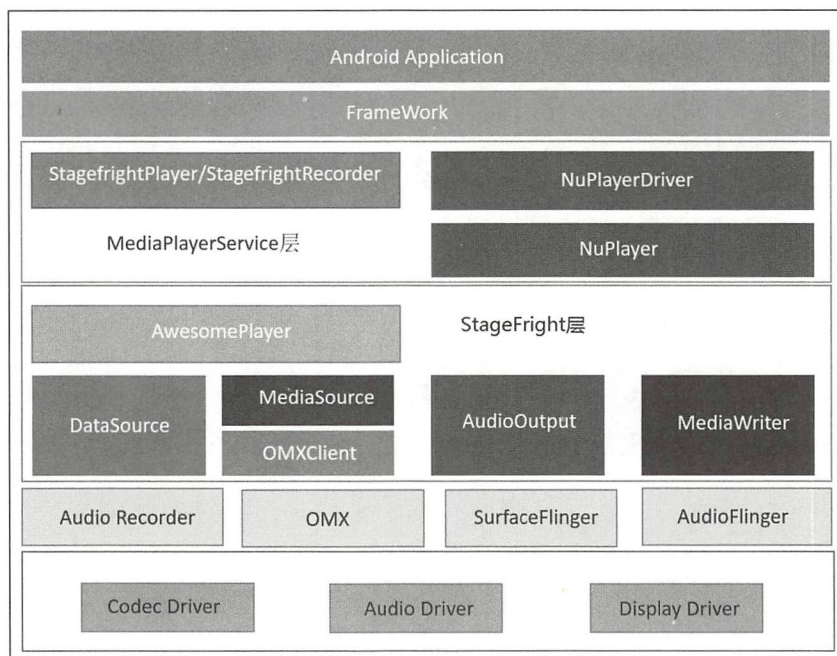


图 3-4 StageFright 层和 MediaPlayerService 层的关系

从图 3-4 中可以看到，StagefrightPlayer/StagefrightRecorder 以及 NuPlayer 都属于 MediaPlayer Service 层。直接到源码中找到 libmediaplayerservice，图 3-5 是 libmediaplayerservice 中的所有文件，其中包括 NuPlayer、MediaPlayerFactory、StagefrightPlayer 等。

图 3-5 中的文件在编译后会生成 libmediaplayerservice.so 文件，当要继续处理从 MediaPlayer 传过来的事件时，就需要到达 libstagefright.so 中，如 AwesomePlayer、MediaExtractor、MediaMuxer、MediaBuffer、MediaSource、MediaCodec、ACodec 等。可以看出这些都是多媒体框架的核心部分，用于数据解析、解码等。举例说明，一个数据源到最终播放的过程，如图 3-6 所示。

经历的过程如下：分离出音视频数据，如果是视频流，通过 OMX 组件解码；如果是音频流，通过音频解码器解码，如 AACDecoder、AC3Decoder 等。解码完成之后，做音视频同步，最终将视频渲染到屏幕上，音频通过音频系统进行播放。



简 > Work (D:) > android-6.0.1_r1 > frameworks > av > media > libmediaplayerservice >			
名称	修改日期	类型	大小
nuplayer	2018/1/22 23:35	文件夹	
tests	2018/1/22 23:35	文件夹	
ActivityManager.cpp	2016/6/14 20:26	CPP 文件	3 KB
ActivityManager.h	2016/6/14 20:26	H 文件	1 KB
Crypto.cpp	2016/6/14 20:26	CPP 文件	8 KB
Crypto.h	2016/6/14 20:26	H 文件	3 KB
Drm.cpp	2016/6/14 20:26	CPP 文件	21 KB
Drm.h	2016/6/14 20:26	H 文件	7 KB
DrmSessionClientInterface.h	2016/6/14 20:26	H 文件	1 KB
DrmSessionManager.cpp	2016/6/14 20:26	CPP 文件	7 KB
DrmSessionManager.h	2016/6/14 20:26	H 文件	3 KB
HDCP.cpp	2016/6/14 20:26	CPP 文件	5 KB
HDCP.h	2016/6/14 20:26	H 文件	2 KB
MediaPlayerFactory.cpp	2016/6/14 20:26	CPP 文件	11 KB
MediaPlayerFactory.h	2016/6/14 20:26	H 文件	4 KB
MediaPlayerService.cpp	2016/6/14 20:26	CPP 文件	72 KB
MediaPlayerService.h	2016/6/14 20:26	H 文件	18 KB
MediaRecorderClient.cpp	2016/6/14 20:26	CPP 文件	9 KB
MediaRecorderClient.h	2016/6/14 20:26	H 文件	4 KB
MetadataRetrieverClient.cpp	2016/6/14 20:26	CPP 文件	10 KB
MetadataRetrieverClient.h	2016/6/14 20:26	H 文件	3 KB
NOTICE	2016/6/14 20:26	文件	11 KB
RemoteDisplay.cpp	2016/6/14 20:26	CPP 文件	2 KB
RemoteDisplay.h	2016/6/14 20:26	H 文件	2 KB
SharedLibrary.cpp	2016/6/14 20:26	CPP 文件	2 KB
SharedLibrary.h	2016/6/14 20:26	H 文件	2 KB
StagefrightPlayer.cpp	2016/6/14 20:26	CPP 文件	6 KB
StagefrightPlayer.h	2016/6/14 20:26	H 文件	3 KB
StagefrightRecorder.cpp	2016/6/14 20:26	CPP 文件	62 KB
StagefrightRecorder.h	2016/6/14 20:26	H 文件	7 KB
TestPlayerStub.cpp	2016/6/14 20:26	CPP 文件	6 KB
TestPlayerStub.h	2016/6/14 20:26	H 文件	5 KB

图 3-5 libmediaplayerservice 的目录结构

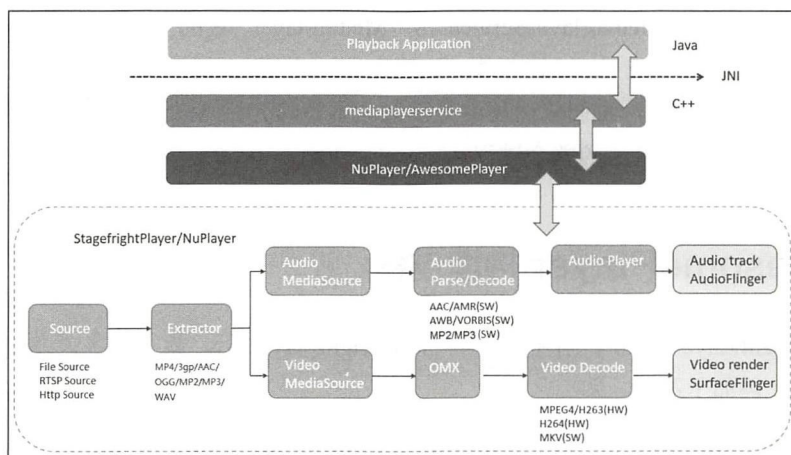


图 3-6 一个数据源到最终播放的过程



第 4 章

StagefrightPlayer (AwesomePlayer)

在第 3 章的最后，MediaPlayerService 中会通过 MediaPlayerFactory 创建两个 PlayerFactory，一个为 StagefrightPlayerFactory，一个为 NuPlayerFactory，前者创建了 StagefrightPlayer（StagefrightPlayer 是对 AwesomePlayer 的封装），后者创建了 NuPlayerDriver（NuPlayerDriver 是对 NuPlayer 的封装）。在 Android 5.1 版本弃用 AwesomePlayer 后，就都使用 NuPlayer 了，但是 Android 5.1 之前的版本还是有 AwesomePlayer 的。本章分析的是 StagefrightPlayer，其也是 AwesomePlayer。

前面章节介绍了 MediaPlayerService 及 MediaPlayer 中的 C/S 模型，创建播放了 StagefrightPlayer，但是对于如何读取数据，如何把数据解析出来，如何渲染到最终的 SurfaceView 上显示并播放，我们依然还不得而知。本章将介绍 StagefrightPlayer 从构造到数据解析，到解码，再到渲染输出的整个过程。

4.1 AwesomePlayer 构造过程

在 StagefrightPlayerFactory 创建了一个 StagefrightPlayer 后，就开始进入 StagefrightPlayer 的构造函数，代码如下：

```
StagefrightPlayer::StagefrightPlayer()  
    : mPlayer(new AwesomePlayer) {  
    mPlayer->setListener(this);  
}
```



IStagefrightPlayer.h 中的声明如下:

```
private:
    AwesomePlayer *mPlayer; //mPlayer 是一个 AwesomePlayer 指针类型的变量
    StagefrightPlayer(const StagefrightPlayer &);
    StagefrightPlayer &operator=(const StagefrightPlayer &);
};
```

从上面的代码中可以看出, StagefrightPlayer 中封装了 AwesomePlayer, 进而进行播放行为相关操作。接着分析一下 StagefrightPlayer, 代码如下:

```
#ifndef ANDROID_STAGEFRIGHTPLAYER_H
#define ANDROID_STAGEFRIGHTPLAYER_H
#include <media/MediaPlayerInterface.h>
namespace android {
struct AwesomePlayer;
class StagefrightPlayer : public MediaPlayerInterface {
public:
    StagefrightPlayer();
    virtual ~StagefrightPlayer();
    virtual status_t initCheck();
    virtual status_t setUID(uid_t uid);
    virtual status_t setDataSource(int fd, int64_t offset, int64_t length);
    virtual status_t setDataSource(const sp<IStreamSource> &source);
    virtual status_t setVideoSurfaceTexture(
        const sp<IGraphicBufferProducer> &bufferProducer);
    virtual status_t prepare();
    virtual status_t prepareAsync();
    virtual status_t start();
    virtual status_t stop();
    virtual status_t pause();
    virtual bool isPlaying();
    virtual status_t seekTo(int msec);
    virtual status_t getCurrentPosition(int *msec);
    virtual status_t getDuration(int *msec);
    virtual status_t reset();
    virtual status_t setLooping(int loop);
    virtual player_type playerType();
    virtual status_t invoke(const Parcel &request, Parcel *reply);
    virtual void setAudioSink(const sp<AudioSink> &audioSink);
    //省略部分代码
private:
    AwesomePlayer *mPlayer;
    StagefrightPlayer(const StagefrightPlayer &);
    StagefrightPlayer &operator=(const StagefrightPlayer &);
};
```




```
};  
} //namespace android  
#endif //ANDROID_STAGEFRIGHTPLAYER_H
```

StagefrightPlayer 继承了 MediaPlayerInterface 抽象基类，这个抽象基类中有很多纯虚函数。

同样以 setDataSource 这个 API 为例，通常当我们负责的模块和别人的模块进行交互通信时，在模块和模块之间，相当于一个黑盒，里面的流程我们并不清楚，调用后，通过返回对应的状态或者数据，可以判断已经发生的事件，进而执行下一个环节。StagefrightPlayer 只是一层外壳，真正干活的是 AwesomePlayer。

在调用 MediaPlayerService 中的 setDataSource 函数后，会到达 StagefrightPlayer 中的 setDataSource 函数：

```
//URI 方式的 setDataSource  
status_t StagefrightPlayer::setDataSource(  
    const sp<IMediaHTTPService> &httpService,  
    const char *url,  
    const KeyedVector<String8, String8> *headers) {  
    return mPlayer->setDataSource(httpService, url, headers);  
}  
//文件类的 setDataSource  
status_t StagefrightPlayer::setDataSource(int fd, int64_t offset, int64_t  
length) {  
    ALOGV("setDataSource(%d, %lld, %lld)", fd, offset, length);  
    return mPlayer->setDataSource(dup(fd), offset, length);  
}  
  
status_t StagefrightPlayer::setDataSource(const sp<IStreamSource> &source) {  
    return mPlayer->setDataSource(source);  
}
```

总结一下上面的代码，所有执行步骤都会调用 mPlayer->setDataSource(xxxx)，而通过前面的分析我们知道，这个 mPlayer 被定义成 AwesomePlayer，虽然 AwesomePlayer 中也有一些网络流处理的代码，但是通常都执行不到这段逻辑，而是使用后面会介绍到的 NuPlayer。接下来到达 AwesomePlayer 的 setDataSource 函数：

```
status_t AwesomePlayer::setDataSource(int fd, int64_t offset, int64_t length) {  
    Mutex::Autolock autoLock(mLock); //互斥锁  
    reset_l();  
    sp<DataSource> dataSource = new FileSource(fd, offset, length);  
    status_t err = dataSource->initCheck(); //检查文件类型  
    mFileSource = dataSource;
```



```
return setDataSource_1(dataSource);
}

status_t AwesomePlayer::setDataSource_1(const sp<DataSource> &dataSource) {
    sp<MediaExtractor> extractor = MediaExtractor::Create(dataSource);
    //数据解析模块
    return setDataSource_1(extractor);
}

status_t AwesomePlayer::setDataSource_1(const sp<MediaExtractor> &extractor) {
    int64_t totalBitRate = 0;
    mExtractor = extractor;
    //遍历数据解析的内容
    for (size_t i = 0; i < extractor->countTracks(); ++i) {
        //得到跟踪的元数据
        sp<MetaData> meta = extractor->getTrackMetaData(i);
        int32_t bitrate;
        if (!meta->findInt32(kKeyBitRate, &bitrate)) {
            const char *mime;
            //检查 mime
            CHECK(meta->findCString(kKeyMIMETYPE, &mime));
            totalBitRate = -1;
            break;
        }
        totalBitRate += bitrate;
        //比特率累加，印证我们常说的比特率越大，文件越大，码率越高，质量也越高
    }
    sp<MetaData> fileMeta = mExtractor->getMetaData();
    if (fileMeta != NULL) {
        int64_t duration;
        if (fileMeta->findInt64(kKeyDuration, &duration)) {
            mDurationUs = duration;
        }
    }
    mBitrate = totalBitRate;
    ALOGV("mBitrate = %lld bits/sec", (long long)mBitrate);
    {
        Mutex::Autolock autoLock(mStatsLock);
        mStats.mBitrate = mBitrate;
        mStats.mTracks.clear();
        mStats.mAudioTrackIndex = -1;
        mStats.mVideoTrackIndex = -1;
    }
    for (size_t i = 0; i < extractor->countTracks(); ++i) {
        sp<MetaData> meta = extractor->getTrackMetaData(i);
```




```
const char *_mime;
CHECK(meta->findCString(kKeyMIMETYPE, &_mime));
String8 mime = String8(_mime);
//分离音视频轨道, 生成 mVideoTrack 和 mAudioTrack 两个 MediaSource
if (!haveVideo && !strncasecmp(mime.string(), "video/", 6)) {
    //与视频相关
    setVideoSource(extractor->getTrack(i));
    haveVideo = true;
    //屏幕上显示区域的宽和高
    int32_t displayWidth, displayHeight;
    bool success = meta->findInt32(kKeyDisplayWidth, &displayWidth);
    if (success) {
        success = meta->findInt32(kKeyDisplayHeight,
                                   &displayHeight);
    }
    if (success) {
        mDisplayWidth = displayWidth;
        mDisplayHeight = displayHeight;
    }
    {
        Mutex::Autolock autoLock(mStatsLock);
        mStats.mVideoTrackIndex = mStats.mTracks.size();
        mStats.mTracks.push();
        TrackStat *stat =
            &mStats.mTracks.editItemAt(mStats.mVideoTrackIndex);
        stat->mMIME = mime.string();
    }
} else if (!haveAudio && !strncasecmp(mime.string(), "audio/", 6)) {
    //如果不是音频, 但却是以 audio/开头的 mime 类型, 如 audio/midi mid midi
    setAudioSource(extractor->getTrack(i));
    haveAudio = true;
    mActiveAudioTrackIndex = i;
    {
        Mutex::Autolock autoLock(mStatsLock);
        mStats.mAudioTrackIndex = mStats.mTracks.size();
        mStats.mTracks.push();
        TrackStat *stat =
            &mStats.mTracks.editItemAt(mStats.mAudioTrackIndex);
        stat->mMIME = mime.string();
    }
    if (!strncasecmp(mime.string(), MEDIA_MIMETYPE_AUDIO_VORBIS)) {
        //如果是 OGG Vorbis, 即 OGG 类型
        sp<MetaData> fileMeta = extractor->getMetaData();
        int32_t loop;
```




```
        if (fileMeta != NULL && fileMeta->findInt32(kKeyAutoLoop,
&loop) && loop != 0) {
            modifyFlags(AUTO_LOOPING, SET);
        }
    }
    } else if (!strcasecmp(mime.string(), MEDIA_MIMETYPE_TEXT_3GPP)) {
//如果是 3GP 类型的
        addTextSource_l(i, extractor->getTrack(i));
        //把解析帧添加到 SurfaceTexture
    }
}

if (!haveAudio && !haveVideo) { //既没有音频流，也没有视频流
//如果有数据解析器，直接解析错误，否则返回错误类型
    if (mWVMEExtractor != NULL) {
        return mWVMEExtractor->getError();
    } else {
        return UNKNOWN_ERROR;
    }
}

mExtractorFlags = extractor->flags();
return OK;
}
```

总结一下上面的代码，AwesomePlayer 主要用于本地播放，匹配不同的文件类型，并用数据解析器处理；mime 表示该资源的媒体类型，当出现常见音视频封装格式的时候，Android 播放器如果在 Manifest.xml 中设置了 mime 对应的类型，就能被播放器识别，如点击手机中的文件图标，就会弹出一个用哪个软件打开的提示。

下面列举一些常用的 mime 类型的资源，如下：

```
video/x-ms-asf asf
video/mpeg mpeg mpg
video/x-msvideo avi
application/vnd.rn-realmedia rm
audio/x-pn-realaudio ram ra
audio/x-aiff aif aiff aifc
audio/mpeg mpga mp3
audio/midi mid midi
audio/wav wav
audio/x-ms-wma wma
video/x-ms-wmv wmv
```



4.2 AwesomePlayer 使用 MediaExtractor 进行数据解析的过程

接着看看上面一直在用的 MediaExtractor（数据解析器）：

```
sp<MetaData> MediaExtractor::getMetaData() {
    return new MetaData;
}

sp<MediaExtractor> MediaExtractor::Create(const sp<DataSource> &source,
const char *mime) {
    sp<AMessage> meta;
    String8 tmp;
    if (mime == NULL) {
        float confidence;
        mime = tmp.string();//获取 mime 值
    }

    bool isDrm = false;
    MediaExtractor *ret = NULL;
    if (!strcasecmp(mime, MEDIA_MIMETYPE_CONTAINER_MPEG4)
        || !strcasecmp(mime, "audio/mp4")) {
        ret = new MPEG4Extractor(source); //MPEG4 类型
    } else if (!strcasecmp(mime, MEDIA_MIMETYPE_AUDIO_MPEG)) {
        ret = new MP3Extractor(source, meta); //MP3 类型
    } else if (!strcasecmp(mime, MEDIA_MIMETYPE_AUDIO_AMR_NB)
        || !strcasecmp(mime, MEDIA_MIMETYPE_AUDIO_AMR_WB)) {
        ret = new AMRExtractor(source);
    } else if (!strcasecmp(mime, MEDIA_MIMETYPE_AUDIO_FLAC)) {
        ret = new FLACExtractor(source);
    } else if (!strcasecmp(mime, MEDIA_MIMETYPE_CONTAINER_WAV)) {
        ret = new WAVExtractor(source);
    } else if (!strcasecmp(mime, MEDIA_MIMETYPE_CONTAINER_OGG)) {
        ret = new OggExtractor(source);
    }
    //省略部分代码
    return ret;
}
```

总结一下上面的代码，针对文件解析的不同格式创建 Extractor 解析器并解析，创建好解析器后回到 AwesomePlayer::setDataSource_1 中，继续执行 setDataSource_1(extractor)函数，对新建的解析器做处理，其实质是做视频的 A（音频）/V（视频）分离。

```
setVideoSource(extractor->getTrack(i)); //设置视频源 mVideoTrack
```



第4章 StagefrightPlayer (AwesomePlayer)

```
setAudioSource(extractor->getTrack(i)); //设置音频源 mAudioTrack
```

mVideoTrack 和 mAudioTrack 作为创建 AwesomePlayer 的成员函数，其类型为 MPEG4Source，继承自 MediaSource：

```
sp<MediaSource> MPEG4Extractor::getTrack(size_t index) {
    status_t err;
    //省略部分代码
    Track *track = mFirstTrack;
    while (index > 0) {
        if (track == NULL) {
            return NULL;
        }
        track = track->next;
        --index;
    }

    Trex *trex = NULL;
    int32_t trackId;
    if (track->meta->findInt32(kKeyTrackID, &trackId)) {
        for (size_t i = 0; i < mTrex.size(); i++) {
            Trex *t = &mTrex.editItemAt(index);
            if (t->track_ID == (uint32_t) trackId) {
                trex = t;
                break;
            }
        }
    }

    return new MPEG4Source(this, track->meta, mDataSource, track ->
timescale, track -> sampleTable, mSidxEntries, trex, mMoofOffset);
}
```

以上过程完成了音视频数据的分离，也就是 demux（解复用），对音频和视频资源分开处理，其顺序如下：MediaPlayerService→StagefrightPlayer→AwesomePlayer→MPEG4Extractor→MPEG4Source。

音频处理过程如图 4-1 所示。

视频处理过程如图 4-2 所示。

AudioPlayer 为 AwesomePlayer 的成员，AudioPlayer 通过 callback 来驱动数据的获取，AwesomePlayer 则通过 videoevent 来驱动。二者有共性，这就是数据的获取都抽象成 mSource->read 来完成，且 read 内部把 parse 和 decode 绑在一起。StageFright A/V 同步部分，Audio 完全由 callback 驱动数据流，注意 Video（视频）部分在 onVideoEvent 里会获取 Audio（音频）的



Android 音视频开发

时间戳，然后进行视频时间与音频时间的比较，计算下一帧间隔多久显示。这样做使时间戳同步。

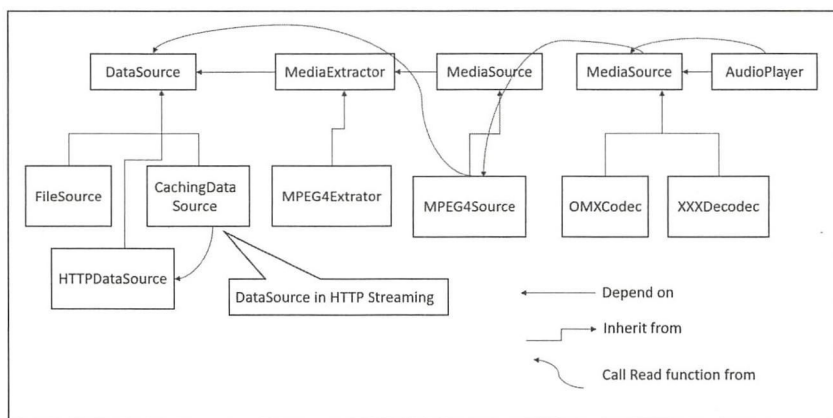


图 4-1 音频处理过程

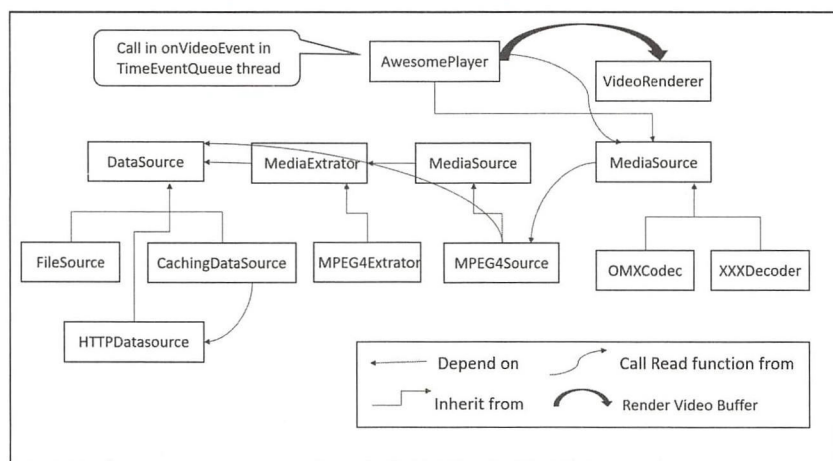


图 4-2 视频处理过程

AwesomePlayer 的 Video 主要有以下几个成员。

- mVideoSource（解码视频）
- mVideoTrack（从多媒体文件中读取视频数据）
- mVideoRenderer（对解码好的视频进行格式转换，Android 使用的格式为 RGB565）
- mISurface（重绘图层）
- mQueue（event 事件队列）



StageFright 运行时的 Audio 流程如下。

- 首先设置 mUri 的路径。
- 启动 mQueue, 创建一个线程来运行 threadEntry (命名为 TimedEventQueue, 这个线程就是 event 调度器)。
- 打开 mUri 所指定文件的头部, 则会根据类型选择不同的分离器 (如 MPEG4Extractor)。
- 使用 MPEG4Extractor 对 MP4 进行音视频轨道的分离, 并返回 MPEG4Source 类型的视频轨道给 mVideoTrack。
- 根据 mVideoTrack 中的编码类型来选择解码器, AVC 的编码类型会选择 AVCD decoder, 返回给 mVideoSource, 并设置 mVideoSource 中的 mSource 为 mVideoTrack。
- 插入 onVideoEvent 到 Queue 中, 开始解码播放。
- 通过 mVideoSource 对象读取解析好的视频 Buffer。

如果解析好的视频 Buffer 还没到 A/V 时间戳同步的时刻, 则推迟到下一轮操作。

(1) mVideoRenderer 为空, 则进行初始化 (如果不使用 OMX, 会将 mVideoRenderer 设置为 AwesomeLocalRenderer)。

(2) 通过 mVideoRenderer 对象将解析好的视频 Buffer 转换成 RGB565 格式, 并发给 display 模块进行图像绘制。

(3) 将 onVideoEvent 重新插入 event 调度器来循环。

4.3 AwesomePlayer 解码过程

4.2 节主要介绍了对本地文件的数据解析, 并分离了音频和视频, 本节将主要介绍 AwesomePlayer 解码过程。

本节将要涉及的知识点如下。

- AwesomePlayer 中的 prepare 过程。
- 初始化音视频解码器过程。
- 使用 OMXCodec 的解码过程。

4.3.1 AwesomePlayer 中的 prepare 过程

首先看看 AwesomePlayer 中的 prepare 过程:

```
status_t AwesomePlayer::prepare() {  
    ATRACE_CALL();
```



```
Mutex::Autolock autoLock(mLock);
return prepare_1();
}

status_t AwesomePlayer::prepare_1() {
    if (mFlags & PREPARED) { // 已经处于 Prepared 状态, 就直接 ok
        return OK;
    }

    if (mFlags & PREPARING) { // 如果是 PREPARING 中, 返回未知错误
        return UNKNOWN_ERROR;
    }

    mIsAsyncPrepare = false;
    status_t err = prepareAsync_1(); // 进入 prepareAsync_1
    while (mFlags & PREPARING) {
        mPreparedCondition.wait(mLock);
    }
    return mPrepareResult;
}

status_t AwesomePlayer::prepareAsync() {
    ATRACE_CALL();
    Mutex::Autolock autoLock(mLock);
    if (mFlags & PREPARING) {
        return UNKNOWN_ERROR;
    }
    // 异步 prepare 已经处于 PREPARING, 再调用 prepareAsync 函数, 返回未知错误
}

mIsAsyncPrepare = true;
return prepareAsync_1();
}

status_t AwesomePlayer::prepareAsync_1() {
    if (mFlags & PREPARING) {
        return UNKNOWN_ERROR;
    }
    // 异步 prepare 已经处于 Prepared 状态, 再调用 prepareAsync 函数, 返回未知错误
}

if (!mQueueStarted) { // TimeEventQueue 未启动, 开启, 并把状态置为 true
    mQueue.start();
    mQueueStarted = true;
}

modifyFlags(PREPARING, SET); // 修改状态为 PREPARING
```




第4章 StagefrightPlayer (AwesomePlayer)

```
//new AwesomeEvent, 并调用 onPrepareAsyncEvent 函数, 返回其状态
mAsyncPrepareEvent = new AwesomeEvent(this, &AwesomePlayer::
onPrepareAsyncEvent);
//post 执行
mQueue.postEvent(mAsyncPrepareEvent);
return OK;
}
```

总结一下上面的代码, prepare 过程调用了 prepareAsync_1 函数, 在 prepareAsync_1 中执行 AwesomeEvent 构造函数, 并将 AwesomePlayer 调用 onPrepareAsyncEvent 的引用结果返回 AwesomeEvent 的构造函数作为参数。

接着分析 AwesomeEvent 的过程。首先启动 mQueue, 作为 EventHandler:

```
struct AwesomeEvent : public TimedEventQueue::Event {
    AwesomeEvent(
        AwesomePlayer *player,
        void (AwesomePlayer::*method)())
        : mPlayer(player),
          mMethod(method) {
    }

protected:
    virtual ~AwesomeEvent() {}
    virtual void fire(TimedEventQueue * /* queue */, int64_t /* now_us */) {
        (mPlayer->*mMethod)();
    }

private:
    AwesomePlayer *mPlayer;
    void (AwesomePlayer::*mMethod)();
    AwesomeEvent(const AwesomeEvent &);
    AwesomeEvent &operator=(const AwesomeEvent &);
};
```

上面的 new AwesomeEvent 会执行 onPrepareAsyncEvent 函数, 下面看看该函数做了什么:

```
void AwesomePlayer::onPrepareAsyncEvent() {
    Mutex::Autolock autoLock(mLock); //互斥锁
    beginPrepareAsync_1(); //开始异步 prepare
}

void AwesomePlayer::beginPrepareAsync_1() {
    if (mFlags & PREPARE_CANCELLED) { //如果放弃 prepare 过程
        ALOGI("prepare was cancelled before doing anything");
    }
}
```



```
        abortPrepare(UNKNOWN_ERROR); //终止 prepare, 返回未知错误的信息
        return;
    }

    if (mUri.size() > 0) { //如果 URI 长度大于 0
        status_t err = finishSetDataSource_l();
        //调用完成 setDataSource 过程, 返回对应的状态
        if (err != OK) {
            abortPrepare(err);
            return;
        }
    }

    if (mVideoTrack != NULL && mVideoSource == NULL) {
        //如果此前视频流 MediaExtrator 中的 mVideoTrack (视频源) 不为 null
        //且 mVideoSource 为空, 初始化视频解码器
        status_t err = initVideoDecoder();
        if (err != OK) {
            abortPrepare(err);
            return;
        }
    }

    if (mAudioTrack != NULL && mAudioSource == NULL) {
        //如果此前音频流 MediaExtrator 中的 mAudioTrack (音频源) 不为 null
        //且 mAudioSource 为空, 初始化音频解码器
        status_t err = initAudioDecoder();
        if (err != OK) {
            abortPrepare(err);
            return;
        }
    }

    modifyFlags(PREPARING_CONNECTED, SET);
    //修改状态为 PREPARING_CONNECTED, 即正在 prepare 的过程
    if (isStreamingHTTP()) { //如果是网络流
        postBufferingEvent_l(); //进行缓冲
    } else {
        finishAsyncPrepare_l(); //文件流直接完成 prepare 过程, 到达 Prepared 状态
    }
}
```

总结一下上面的代码, 会将 A/V (音视频) 分别处理, 于是有了 `AwesomePlayer::initVideoDecoder` 及 `AwesomePlayer::initAudioDecoder` 函数。



4.3.2 初始化音视频解码器过程

下面先看看 initVideoDecoder，即初始化视频解码器：

```
status_t AwesomePlayer::initVideoDecoder(uint32_t flags) {
    ATRACE_CALL();
    ALOGV("initVideoDecoder flags=0x%x", flags);
    //Codec 就是音频编解码器，主要负责编码、解码相关操作
    //其中有几个参数：mVideoTrack-getFormat 用于获取视频轨道的格式
    //mVideoTrack 是一个MediaSource，即视频轨道
    mVideoSource = OMXCodec::Create(
        mClient.interface(), mVideoTrack->getFormat(),
        false, // createEncoder
        mVideoTrack,
        NULL, flags, USE_SURFACE_ALLOC ? mNativeWindow : NULL);

    if (mVideoSource != NULL) {
        int64_t durationUs;
        if (mVideoTrack->getFormat()->findInt64(kKeyDuration, &durationUs)) {
            Mutex::Autolock autoLock(mMiscStateLock);
            if (mDurationUs < 0 || durationUs > mDurationUs) {
                mDurationUs = durationUs;
            }
        }

        status_t err = mVideoSource->start(); //通过 read 函数读取原始视频数据
                                              //省略读取异常处理的代码
    }

    if (mVideoSource != NULL) {
        const char *componentName;
        {
            Mutex::Autolock autoLock(mStatsLock);
            TrackStat *stat = &mStats.mTracks.editItemAt(mStats.
mVideoTrackIndex);
            stat->mDecoderName = componentName;
        }

        static const char *kPrefix = "OMX.Nvidia.";
        static const char *kSuffix = ".decode";
        static const size_t kSuffixLength = strlen(kSuffix);
        size_t componentNameLength = strlen(componentName);
        if (!strncmp(componentName, kPrefix, strlen(kPrefix))
            && componentNameLength >= kSuffixLength
            && !strcmp(&componentName[
```



```

        componentNameLength - kSuffixLength], kSuffix)) {
            modifyFlags(SLOW_DECODER_HACK, SET);
        }
    }
    return mVideoSource != NULL ? OK : UNKNOWN_ERROR;
}

```

接着看看初始化音频解码器，其中有几个变量的声明：

```

sp<MediaSource> mAudioTrack; //音频 Track
sp<MediaSource> mAudioSource; //分离出的音频 Source
sp<AwesomeRenderer> mVideoRenderer; //视频渲染
sp<MediaSource> mVideoTrack; //视频 Track
sp<MediaSource> mVideoSource; //分离出的视频 Source

```

接着看看相关代码：

```

status_t AwesomePlayer::initAudioDecoder() {
    ATRACE_CALL();
    sp<MetaData> meta = mAudioTrack->getFormat(); //得到音频轨道跟踪的格式
    //省略部分代码
    audio_stream_type_t streamType = AUDIO_STREAM_MUSIC;
    if (mAudioSink != NULL) {
        streamType = mAudioSink->getAudioStreamType(); //得到音频流类型
    }
    mOffloadAudio = canOffloadStream(meta, (mVideoSource != NULL),
                                     isStreamingHTTP(), streamType);
    if (!strcasecmp(mime, MEDIA_MIMETYPE_AUDIO_RAW)) {
        ALOGV("createAudioPlayer: bypass OMX (raw)");
        mAudioSource = mAudioTrack;
    } else {
        //这里几个参数，除了 mVideoTrack 变成 mAudioTrack，其他与视频流相关代码一样
        mOmxSource = OMXCodec::Create(//创建 OMXCodec
                                     mClient.interface(), mAudioTrack->getFormat(),
                                     false, // createEncoder
                                     mAudioTrack);
    }
    //省略部分代码
    if (mAudioSource != NULL) {
        int64_t durationUs;
        if (mAudioTrack->getFormat()->findInt64(kKeyDuration, &durationUs)) {
            Mutex::Autolock autoLock(mMiscStateLock);
            if (mDurationUs < 0 || durationUs > mDurationUs) {
                mDurationUs = durationUs;
            }
        }
    }
}

```

```

        status_t err = mAudioSource->start(); //读取原始音频数据
        if (err != OK) {
            mAudioSource.clear();
            mOmxSource.clear();
            return err;
        }
    } else if (!strcascmp(mime, MEDIA_MIMETYPE_AUDIO_QCELP)) {
        return OK;
    }
    return mAudioSource != NULL ? OK : UNKNOWN_ERROR;
}

```

总结一下上面的代码，在 StageFright 调用 AwesomePlayer 的 prepare 函数后，AwesomePlayer 调用自身的 prepareAsync 初始化音视频解码器，无论是 prepare 函数还是 prepareAsync 函数，都会触发 OMXCodec::Create 函数，然后使用 OMXCodec 处理对应的业务逻辑。prepareAsync 阶段主要完成以下 3 件事。

- (1) Streaming: 启动下载数据并缓存。
- (2) 初始化并创建音视频解码器。
- (3) 通知上层，已经处于 Prepared 状态。

4.3.3 使用 OMXCodec 的解码过程

经过“数据流的封装”得到的两个 MediaSource，其实是两个 OMXCodec。AwesomePlayer 和 mAudioPlayer 都是从 MediaSource 中得到数据并进行播放的。AwesomePlayer 得到的是最终需要渲染的原始视频数据，而 mAudioPlayer 得到的是最终需要播放的原始音频数据。也就是说，从 OMXCodec 中读到的数据已经是原始数据了。

OMXCodec 是怎么把数据源经过 parse、decode 两步以后转化成原始数据的呢？从 OMXCodec::Create 这个构造函数开始，下面看看它的代码，位于 frameworks\av\media\libstagefright\OMXCodec.cpp 文件中：

```

sp<MediaSource> OMXCodec::Create(
    const sp<IOmx> &omx,
    const sp<MetaData> &meta, bool createEncoder,
    const sp<MediaSource> &source,
    const char *matchComponentName,
    uint32_t flags,
    const sp<ANativeWindow> &nativeWindow) {
    int32_t requiresSecureBuffers;

```



```

    const char *mime;
    bool success = meta->findCString(kKeyMIMETYPE, &mime);
    CHECK(success);
    Vector<CodecNameAndQuirks> matchingCodecs;
    findMatchingCodecs(mime, createEncoder, matchComponentName, flags,
&matchingCodecs);
    if (matchingCodecs.isEmpty()) {
        return NULL;
    }

    sp<OMXCodecObserver> observer = new OMXCodecObserver;
    IOMX::node_id node = 0;
    for (size_t i = 0; i < matchingCodecs.size(); ++i) {
        const char *componentNameBase = matchingCodecs[i].mName.string();
        uint32_t quirks = matchingCodecs[i].mQuirks;
        const char *componentName = componentNameBase;
        AString tmp;
        if (flags & kUseSecureInputBuffers) {
            tmp = componentNameBase;
            tmp.append(".secure");
            componentName = tmp.c_str();
        }
        if (createEncoder) { //创建编码器
            sp<MediaSource> softwareCodec =
                InstantiateSoftwareEncoder(componentName, source, meta);
            //实例化软件编码器
            if (softwareCodec != NULL) {
                ALOGV("Successfully allocated software codec '%s'",
componentName);
                return softwareCodec;
            }
        }
        if (!createEncoder && (quirks & kOutputBuffersAreUnreadable)
            && (flags & kClientNeedsFramebuffer)) {
            if (strncmp(componentName, "OMX.SEC.", 8)) {
                //比较组件名是否为 OMX.SEC
                continue;
            }
        }
        status_t err = omx->allocateNode(componentName, observer, &node);
        if (err == OK) {
            ALOGV("Successfully allocated OMX node '%s'", componentName);
            sp<OMXCodec> codec = new OMXCodec(
                omx, node, quirks, flags,

```



```

        createEncoder, mime, componentName,
        source, nativeWindow);
    observer->setCodec(codec);
    err = codec->configureCodec(meta);
    if (err == OK) {
        return codec;
    }
    ALOGV("Failed to configure codec '%s'", componentName);
}
}
return NULL;
}

```

总结一下上面的代码，对应参数分析如下。

- IOMX &omx 指的是一个 OMXNodeInstance 对象的实例。
- MetaData &meta 这个参数由 MediaSource.getFormat 获取得到。这个对象的主要成员就是一个 KeyedVector(uint32_t, typed_data) mItems，里面存放了一些代表 MediaSource 格式信息的键/值对。
- bool createEncoder 指明这个 OMXCodec 是编码还是解码。
- MediaSource &source 是一个 MediaExtractor（数据解析器）。
- char *matchComponentName 指定一种 Codec 用于生成这个 OMXCodec。

先使用 findMatchingCodecs 寻找对应的 Codec，找到以后为当前 IOMX 分配节点并注册事件监听器：omx->allocateNode(componentName, observer, &node)。最后，把 IOMX 封装进一个 OMXCodec：

```

sp<OMXCodec> codec = new OMXCodec(
    omx, node, quirks,
    createEncoder, mime, componentName,
    source);

```

这样就得到了 OMXCodec。

在 AwesomePlayer 中得到这个 OMXCodec 后，接着看看 initVideoDecoder/initAudioDecoder，此处看看 initAudioDecoder 函数，使 mAudioSource = mOmxCSource，接着调用 mAudioSource->start 进行初始化。OMXCodec 初始化主要做如下两件事。

- 向 OpenMAX 发送开始指令，如 mOMX->sendCommand(mNode, OMX_CommandStateSet, OMX_StateIdle)，表示状态就绪了，可以进入预解码阶段。
- 调用 allocateBuffers 函数分配两个缓冲区，存放在变量 mPortBuffers[2]中，分别用于输入和输出。

然后在 `initxxxDecoder`（初始化音频/视频）函数中调用（`mAudioSource->start/mVideoSource->start`）:

```
struct MediaSource : public virtual RefBase {
    MediaSource();
    //省略部分代码
    virtual status_t start(MetaData *params = NULL) = 0;
    virtual status_t stop() = 0;
    //返回 MediaSource 数据格式
    virtual sp<MetaData> getFormat() = 0;
    //省略部分代码
protected:
    virtual ~MediaSource();

private:
    MediaSource(const MediaSource &);
    MediaSource &operator=(const MediaSource &);
};
```

触发 `MediaSource` 的子类 `VideoSource` 及 `AudioSource` 调用 `start` 函数后，它的内部就会开始从数据源获取数据并解析，等到缓冲区满后便停止。在 `AwesomePlayer` 里就可以调用 `MediaSource` 的 `read` 函数读取解码后的数据。

- 对于 `mVideoSource` 来说，将读取的数据（`mVideoSource->read(&mVideoBuffer, &options)`）交给显示模块进行渲染（`mVideoRenderer->render(mVideoBuffer)`）。
- 对 `mAudioSource` 来说，用 `mAudioPlayer` 对 `mAudioSource` 进行封装，然后由 `mAudioPlayer` 负责读取数据和控制播放。
- `AwesomePlayer` 调用 `OMXCodec` 读取 ES（Elementary Stream，也叫基本码流，包含视频、音频或数据的连续码流），并且进行解码处理。
- `OMXCodec` 调用 `MediaSource` 的 `read` 函数来获取音视频数据。
- `OMXCodec` 调用 Android 的 `IOMX` 接口来实现一些音视频解码操作。

这个过程就是 `prepare` 过程，主要用于分离音视频数据，给音频和视频申请好 `Buffer` 空间，做预解码操作。

接下来，当 Java 层调用 `start` 函数时，通过 `MediaPlayerService` 进行 IPC 通信后，再调用 `StagefrightPlayer` 中的 `start` 函数，引用 `AwesomePlayer`，这样就调用了 `AwesomePlayer` 的 `play` 函数，下面查看代码：

```
status_t AwesomePlayer::play() {
    ATRACE_CALL();
    Mutex::Autolock autoLock(mLock);
```

```

        modifyFlags(CACHE_UNDERRUN, CLEAR);
        return play_l();
    }

    status_t AwesomePlayer::play_l() {
        if (mAudioSource != NULL) {
            if (mAudioPlayer == NULL) {
                createAudioPlayer_l();
            }
            if (mVideoSource == NULL) {
                status_t err = startAudioPlayer_l(
                    false /* sendErrorNotification */);
                if ((err != OK) && mOffloadAudio) {
                    ALOGI("play_l() cannot create offload output, fallback
to sw decode");

                    int64_t curTimeUs;
                    getPosition(&curTimeUs);
                    delete mAudioPlayer;
                    mAudioPlayer = NULL;
                    //如果播放器已经处于 Started 状态了, 在 Destroyed 状态时, 需要调用 stop 来结束
                    if (!(mFlags & AUDIOPLAYER_STARTED)) {
                        mAudioSource->stop();
                    }
                    modifyFlags((AUDIO_RUNNING | AUDIOPLAYER_STARTED), CLEAR);
                    mOffloadAudio = false;
                    mAudioSource = mOmxSource;
                    if (mAudioSource != NULL) {
                        err = mAudioSource->start();
                        if (err != OK) {
                            mAudioSource.clear();
                        } else {
                            mSeekNotificationSent = true;
                            if (mExtractorFlags & MediaExtractor::CAN_SEEK) {
                                seekTo_l(curTimeUs);
                            }
                        }
                        createAudioPlayer_l();
                        err = startAudioPlayer_l(false);
                    }
                }
            }
        }
        return OK;
    }
}

```

在 AwesomePlayer 调用 play 后, 通过 mVideoSource->read(&mVideoBuffer, &options)读取

数据。mVideoSource->read(&mVideoBuffer, &options)具体是通过调用 OMXCodec.read 来读取数据的。而 OMXCodec.read 主要分如下两步来实现数据的读取。

- 通过调用 drainInputBuffers 函数对 mPortBuffers[kPortIndexInput]进行填充，这一步完成 parse。由 OpenMAX 从数据源把 demux（解复用）后的数据读取到输入缓冲区，作为 OpenMAX 的输入。
- 通过 fillOutputBuffers 函数对 mPortBuffers[kPortIndexOutput]进行填充，这一步完成解码数据的填充。由 OpenMAX 对输入缓冲区中的数据进行解码，然后把解码后可以显示的视频数据输出到输出缓冲区。

AwesomePlayer 通过 mVideoRenderer->render(mVideoBuffer)对经过 parse 和 decode 处理的数据进行渲染。一个 mVideoRenderer 其实就是一个包装了 IOMXRenderer 的 AwesomeRemoteRenderer:

```
mVideoRenderer = new AwesomeRemoteRenderer(  
    mClient.interface()->createRenderer(  
        mISurface, component,  
        (OMX_COLOR_FORMATTYPE) format,  
        decodedWidth, decodedHeight,  
        mVideoWidth, mVideoHeight,  
        rotationDegrees));
```

4.4 AwesomePlayer 的渲染输出过程

4.3 节最后介绍到数据解码后放到 Buffer 的过程，而本节将分析 AwesomePlayer 音视频输出过程。

本节将要涉及的知识点如下。

- 用一张图回顾数据处理过程。
- 视频渲染器构建过程。
- 将音频数据放到 Buffer 的过程。
- AudioPlayer 在 AwesomePlayer 运行的过程。
- 音视频同步。
- 音视频输出。

4.4.1 用一张图回顾数据处理过程

解码组件 OMXCodec 读取 MediaSource 中的数据进行解码，VideoRender 读取解码后的视

第4章 StagefrightPlayer (AwesomePlayer)

频数据进行渲染，AudioPlayer 获取解码后的音频数据进行播放。图 4-3 就是数据处理过程的图形表示。

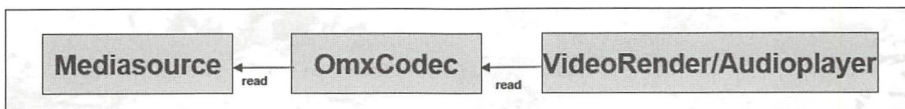


图 4-3 数据处理过程的图形表示

4.4.2 视频渲染器构建过程

视频渲染器的构建过程，是在通知解码完成的事件回调函数中，代码如下：

```

AwesomePlayer::AwesomePlayer()
//省略部分代码
    mVideoEvent = new AwesomeEvent(this, &AwesomePlayer::onVideoEvent);
}

```

在 AwesomePlayer 构造函数中构造一个 AwesomeEvent 时，传入 AwesomePlayer::onVideoEvent 的引用地址，代码如下：

```

void AwesomePlayer::onVideoEvent() {
    ATRACE_CALL();
    Mutex::Autolock autoLock(mLock);
    mVideoEventPending = false;
    if (mSeeking != NO_SEEK) { //不是 SEEK
        if (mVideoBuffer) {
            mVideoBuffer->release();
            mVideoBuffer = NULL;
        }
        if (mSeeking == SEEK && isStreamingHTTP() && mAudioSource != NULL
        && !(mFlags & SEEK_PREVIEW)) { //是 SEEK 且以 HTTP 流的形式快进/快退视频源时，
            //首先应跟随其音频源。为避免 SEEK 时跳跃，只要视频解码已经从新的位置读数据，
            //直到下一个快进/快退，这期间可以暂停音频源播放
            if (mAudioPlayer != NULL && (mFlags & AUDIO_RUNNING)) {
                mAudioPlayer->pause(); //pause
                modifyFlags(AUDIO_RUNNING, CLEAR); //CLEAR 音频正在播放
            }
            mAudioSource->pause();
        }
    }

    if (!mVideoBuffer) {
        MediaSource::ReadOptions options;
        if (mSeeking != NO_SEEK) {

```

Android 音视频开发

```

        ALOGV("seeking to %" PRId64 " us (%.2f secs)", mSeekTimeUs,
mSeekTimeUs / 1E6);
        options.setSeekTo(
            mSeekTimeUs,
            mSeeking == SEEK_VIDEO_ONLY
                ? MediaSource::ReadOptions::SEEK_NEXT_SYNC
                : MediaSource::ReadOptions::SEEK_CLOSEST_SYNC);
    }
    for (;;) {
        //读取里面的数据
        status_t err = mVideoSource->read(&mVideoBuffer, &options);
        options.clearSeekTo();
        if (err != OK) {
            CHECK(mVideoBuffer == NULL);
            if (err == INFO_FORMAT_CHANGED) {
                ALOGV("VideoSource signalled format change.");
                notifyVideoSize_l();
                if (mVideoRenderer != NULL) { //如果渲染器不为空
                    mVideoRendererIsPreview = false;
                    initRenderer_l(); //初始化渲染器
                }
                continue;
            }
        }
        //video playback 完成后，我们也许依然有一个 seek 请求延迟，需要应用于音频中
        finishSeekIfNecessary(-1);
        if (mAudioPlayer != NULL
            && !(mFlags & (AUDIO_RUNNING | SEEK_PREVIEW))) {
            startAudioPlayer_l(); //开始音频解码
        }
        modifyFlags(VIDEO_AT_EOS, SET);
        postStreamDoneEvent_l(err);
        return;
    }
    //VideoBuffer 表示视频的 Buffer
    if (mVideoBuffer->range_length() == 0) {
        //值得注意的是，PV AVC 软解码器返回空的 Buffer，我们可以忽略
        mVideoBuffer->release();
        mVideoBuffer = NULL;
        continue;
    }
    break;
}
{
    Mutex::Autolock autoLock(mStatsLock);

```



```

        ++mStats.mNumVideoFramesDecoded;
    }
}
int64_t timeUs;
CHECK(mVideoBuffer->meta_data()->findInt64(kKeyTime, &timeUs));
//重点语句, 先标记
mLastVideoTimeUs = timeUs;
{
    Mutex::Autolock autoLock(mMiscStateLock);
    mVideoTimeUs = timeUs;
}

SeekType wasSeeking = mSeeking;
finishSeekIfNecessary(timeUs);
if (mAudioPlayer != NULL && !(mFlags & (AUDIO_RUNNING | SEEK_PREVIEW))) {
    status_t err = startAudioPlayer_1();
    //省略部分代码
}

if ((mFlags & TEXTPLAYER_INITIALIZED)
    && !(mFlags & (TEXT_RUNNING | SEEK_PREVIEW))) {
    mTextDriver->start();
    modifyFlags(TEXT_RUNNING, SET);
}

TimeSource *ts =
    ((mFlags & AUDIO_AT_EOS) || !(mFlags & AUDIOPLAYER_STARTED))
        ? &mSystemTimeSource : mTimeSource;
int64_t systemTimeUs = mSystemTimeSource.getRealTimeUs();
int64_t looperTimeUs = ALooper::GetNowUs();
if (mFlags & FIRST_FRAME) {
    modifyFlags(FIRST_FRAME, CLEAR);
    mSinceLastDropped = 0;
    mClockEstimator->reset();
    mTimeSourceDeltaUs = estimateRealTimeUs(ts, systemTimeUs) - timeUs;
}

int64_t realTimeUs, mediaTimeUs;
if (!(mFlags & AUDIO_AT_EOS) && mAudioPlayer != NULL && mAudioPlayer
-> getMediaTimeMapping(&realTimeUs, &mediaTimeUs)) {
    ALOGV("updating TSdelta (%" PRId64 " => %" PRId64 " change %"
PRId64 ")", mTimeSourceDeltaUs, realTimeUs - mediaTimeUs, mTimeSourceDeltaUs
- (realTimeUs - mediaTimeUs));
    ATRACE_INT("TS delta change (ms)", (mTimeSourceDeltaUs -

```

Android 音视频开发

```

(realTimeUs - mediaTimeUs) / 1E3);
    mTimeSourceDeltaUs = realTimeUs - mediaTimeUs;
}

    if ((mNativeWindow != NULL) && (mVideoRendererIsPreview || mVideoRenderer
== NULL)) {
        mVideoRendererIsPreview = false;
        initRenderer_1();
    }

    if (mVideoRenderer != NULL) {
        mSinceLastDropped++;
        mVideoBuffer->meta_data()->setInt64(kKeyTime, loopTimeUs -
latenessUs);
        mVideoRenderer->render(mVideoBuffer);
        if (!mVideoRenderingStarted) {
            mVideoRenderingStarted = true;
            notifyListener_1(MEDIA_INFO, MEDIA_INFO_RENDERING_START);
        }

        if (mFlags & PLAYING) {
            notifyIfMediaStarted_1();
        }
    }

    mVideoBuffer->release();
    mVideoBuffer = NULL;
    if (wasSeeking != NO_SEEK && (mFlags & SEEK_PREVIEW)) {
        modifyFlags(SEEK_PREVIEW, CLEAR);
        return;
    }
    postVideoEvent_1();
}

```

以上代码最后会调用 `initRenderer_1` 函数，代码如下：

```

void AwesomePlayer::initRenderer_1() {
    ATRACE_CALL();
    if (mNativeWindow == NULL) {
        return;
    }
    sp<MetaData> meta = mVideoSource->getFormat();
    int32_t format;
    const char *component;
    int32_t decodedWidth, decodedHeight;

```


第4章 StagefrightPlayer (AwesomePlayer)

```

//省略部分代码
int32_t rotationDegrees;
if (!mVideoTrack->getFormat()->findInt32(
    kKeyRotation, &rotationDegrees)) {
    rotationDegrees = 0;
}
mVideoRenderer.clear();
IPCThreadState::self()->flushCommands();
//在设置缩放模式失败后,持续执行后面的过程,不会受当前状态的影响
setVideoScalingMode_1(mVideoScalingMode);
if (USE_SURFACE_ALLOC
    && !strcmp(component, "OMX.", 4)
    && strcmp(component, "OMX.google.", 11)) {
    //通过直接获取 ANativeBuffers 这一 Buffer,硬解码避免了 CPU 的颜色转
    //换,所以必须用一个渲染器,仅仅是把这些 Buffer 渲染到 ANativeWindow 上
    mVideoRenderer = new AwesomeNativeWindowRenderer(mNativeWindow,
rotationDegrees);
} else {
    //当其他解码器在 Native 中被实例化时,分配的 Buffer 是在 Native 的地址空间,
    //渲染器执行颜色转换并且将复制得到数据 (ANativeBuffer) 渲染到 ANativeWindow 上
    sp<AMessage> format;
    convertMetaDataToMessage(meta, &format);
    mVideoRenderer = new AwesomeLocalRenderer(mNativeWindow, format);
}
}

```

总结一下上面的代码, AwesomeRemoteRenderer 的本质是由 OMX::createRenderer 建立一个硬件渲染器 (hardware renderer), 逻辑为如果是硬解码, 则构建 AwesomeNativeWindowRenderer 渲染器; 如果是软解码, 则构建 AwesomeLocalRenderer 渲染器。

接下来看看两个渲染器有什么区别? 首先是 AwesomeNativeWindowRenderer 渲染器:

```

struct AwesomeNativeWindowRenderer : public AwesomeRenderer {
    AwesomeNativeWindowRenderer(
        const sp<ANativeWindow> &nativeWindow,
        int32_t rotationDegrees)
        : mNativeWindow(nativeWindow) {
        applyRotation(rotationDegrees);
    }

    virtual void render(MediaBuffer *buffer) {
        ATRACE_CALL();
        int64_t timeUs;
        CHECK(buffer->meta_data()->findInt64(kKeyTime, &timeUs));
        native_window_set_buffers_timestamp(mNativeWindow.get(), timeUs

```


Android 音视频开发

```

* 1000);

    status_t err = mNativeWindow->queueBuffer(
        mNativeWindow.get(), buffer->graphicBuffer().get(), -1);
    //省略部分代码
    sp<MetaData> metaData = buffer->meta_data();
    metaData->setInt32(kKeyRendered, 1);
}

protected:
    virtual ~AwesomeNativeWindowRenderer() {}
private:
    sp<ANativeWindow> mNativeWindow;
    //省略部分代码
    AwesomeNativeWindowRenderer(const AwesomeNativeWindowRenderer &);
    AwesomeNativeWindowRenderer &operator=(
        const AwesomeNativeWindowRenderer &);
};

```

以上代码最重要的是 `render` 函数，把要渲染的数据入队到 `NativeWindow`（本地服务窗口），通过 `NativeWindow` 来渲染数据，而另一个 `AwesomeLocalRenderer` 渲染器主要用于软解码时进行渲染。从下面代码中的一些变量的初始化操作可以看出：

```

struct AwesomeLocalRenderer : public AwesomeRenderer {
    AwesomeLocalRenderer( const sp<ANativeWindow> &nativeWindow, const
        sp<AMessage> &format) : mFormat(format),
        mTarget(new SoftwareRenderer (nativeWindow)) {
    }

    virtual void render(MediaBuffer *buffer) {
        int64_t timeUs;
        //省略部分代码
        render((const uint8_t *)buffer->data() + buffer->range_offset(),
            buffer->range_length(), timeUs, timeUs * 1000);
    }

    void render(const void *data, size_t size, int64_t mediaTimeUs, nsecs_t
renderTimeNs) {
        (void)mTarget->render(data, size, mediaTimeUs, renderTimeNs, NULL,
mFormat);
    }

protected:
    virtual ~AwesomeLocalRenderer() {
        delete mTarget;
        mTarget = NULL;
    }
};

```

```

    }

private:
    sp<AMessage> mFormat;
    SoftwareRenderer *mTarget;
    AwesomeLocalRenderer(const AwesomeLocalRenderer &);
    AwesomeLocalRenderer &operator=(const AwesomeLocalRenderer &);
};

```

AwesomeLocalRenderer 的本质是由 OMX 开始创建的 Renderer，createRenderer 函数用于建立一个渲染器。如果 video decoder 是 software component，则建立一个 AwesomeLocalRenderer 作为 mVideoRenderer。

AwesomeLocalRenderer 的构造函数会呼叫本身的 init 函数，其所做的事和 OMX::createRenderer 一模一样，可以理解为把读取到的数据显示在渲染器中。

在渲染器渲染出画面后，我们可能会想，MediaExtractor 把音频和视频分开了，那么之后怎么让解码后的音视频数据保持同步呢？这在之后的章节中会进行介绍。

4.4.3 将音频数据放到 Buffer 的过程

无论是音频，还是视频，都是 bufferdata，音频或视频总有一个维持时间线的流。举一个例子，我们都看过双簧表演，一个人说话，一个人演示动作，动作快了不行，话说快了而动作跟不上也不行，正式表演前练习的时候，自然会设置一些停顿或暗号，使两人的声音和动作保持同步。音频和视频的同步与此类似，在 OpenCore 框架中，设置了一个主时钟，而音频和视频就分别参考主时钟作为输出的依据。而在 StageFright 框架中，音频的输出是通过回调函数来驱动的，视频则根据音频的时间戳来做同步。

为了更好地理解音视频同步，下面先了解一下音频相关播放过程。

在 StageFright 框架中，音频部分是交由 AudioPlayer 来处理的，它是在 AwesomePlayer 的 play_l 函数中被构建的，代码如下：

```

status_t AwesomePlayer::play_l() {
    if (mAudioSource != NULL) {
        if (mAudioPlayer == NULL) {
            createAudioPlayer_l(); // 这里创建 AudioPlayer
        }

        CHECK(!(mFlags & AUDIO_RUNNING));
        if (mVideoSource == NULL) { // 视频源为空
            status_t err = startAudioPlayer_l(
                false /* sendErrorNotification */);

```


Android 音视频开发

```

        if ((err != OK) && mOffloadAudio) {
            ALOGI("play_l() cannot create offload output, fallback
to sw decode");

            int64_t curTimeUs;
            getPosition(&curTimeUs); //得到当前音频的时间戳位置
            delete mAudioPlayer; //释放 AudioPlayer
            mAudioPlayer = NULL;
            //当音频源为空，当前音频播放已经处于 Started 状态时，需要在销毁时，调用 stop 函数
            if (!(mFlags & AUDIOPLAYER_STARTED)) {
                mAudioSource->stop();
            }
            modifyFlags((AUDIO_RUNNING | AUDIOPLAYER_STARTED), CLEAR);
            mOffloadAudio = false;
            mAudioSource = mOmxSource; //清掉之前的状态，即将播放音频源
            if (mAudioSource != NULL) {
                err = mAudioSource->start();
                if (err != OK) {
                    mAudioSource.clear();
                } else {
                    mSeekNotificationSent = true;
                    if (mExtractorFlags & MediaExtractor::CAN_SEEK) {
                        seekTo_l(curTimeUs); //快进/快退到当前的时间戳
                    }
                    createAudioPlayer_l();
                    err = startAudioPlayer_l(false);
                }
            }
        }

        if (mAudioPlayer != NULL) {
            mAudioPlayer->setPlaybackRate(mPlaybackSettings);
        }

        if (mTimeSource == NULL && mAudioPlayer == NULL) {
            mTimeSource = &mSystemTimeSource;
        }
    }
}

```

下面创建 AudioPlayer:

```

void AwesomePlayer::createAudioPlayer_l()
{
    //省略部分代码
}

```



```

mAudioPlayer = new AudioPlayer(mAudioSink, flags, this);
mAudioPlayer->setSource(mAudioSource); //发送数据给音频播放器
//省略部分代码
}

```

接着看看 startAudioPlayer_l 函数:

```

status_t AwesomePlayer::startAudioPlayer_l(bool sendErrorNotification) {
    //省略部分代码
    if (!(mFlags & AUDIOPLAYER_STARTED)) {
        bool wasSeeking = mAudioPlayer->isSeeking();
        //为了能让解码器预读取那些已经分离出来的音频数据, 传入参数 true
        err = mAudioPlayer->start(true /* sourceAlreadyStarted */);
        modifyFlags(AUDIOPLAYER_STARTED, SET);
        if (wasSeeking) {
            CHECK(!mAudioPlayer->isSeeking());
            //在启动音频播放后, 需要结束快进/快退到对应时间点的操作
            postAudioSeekComplete();
        } else {
            notifyIfMediaStarted_l();
            //没有快进/快退到指定点状态, 直接向外通知音频正在解码
        }
    } else {
        err = mAudioPlayer->resume();
    }
    if (err == OK) {
        err = mAudioPlayer->setPlaybackRate(mPlaybackSettings);
    }
    if (err == OK) {
        modifyFlags(AUDIO_RUNNING, SET);
        mWatchForAudioEOS = true;
    }
    return err;
}

```

接下来看看音频 mAudioPlayer->start(true)的操作, 上面的过程都是在 AwesomePlayer 中发生的, 接下来追踪到 AudioPlayer.cpp 类中:

```

status_t AudioPlayer::start(bool sourceAlreadyStarted) {
    //省略部分代码
    MediaSource::ReadOptions options;
    if (mSeeking) { //如果有快进/快退操作, 就通过读取参数快进/快退到指定的时间戳
        options.setSeekTo(mSeekTimeUs);
        mSeeking = false;
    }
    //读取第一部分 (就是之前音视频分离的音频部分, 这是因为传入的参数是 true)
}

```

```

mFirstBufferResult = mSource->read(&mFirstBuffer, &options);
sp<MetaData> format = mSource->getFormat();//得到音频源格式
//省略部分代码
audio_format_t audioFormat = AUDIO_FORMAT_PCM_16_BIT;
int avgBitRate = -1;
format->fidInt32(kKeyBitRate, &avgBitRate);
if (mAudioSink.get() != NULL) {
    status_t err = mAudioSink->open(
        mSampleRate, numChannels, channelMask, audioFormat,
        DEFAULT_AUDIOSINK_BUFFERCOUNT,
        &AudioPlayer::AudioSinkCallback,
        this,
        (audio_output_flags_t)flags,
        useOffload() ? &offloadInfo : NULL);
    if (err == OK) {
        mLatencyUs = (int64_t)mAudioSink->latency() * 1000;
        mFrameSize = mAudioSink->frameSize();
        //省略部分代码
        err = mAudioSink->start();
        if (!useOffload()) {
            err = OK;
        }
    }
} else {
    //通过构建 AudioTrack 来播放音轨
    mAudioTrack = new AudioTrack(
        AUDIO_STREAM_MUSIC, mSampleRate, AUDIO_FORMAT_PCM_16_BIT,
        audioMask, 0 /*frameCount*/, AUDIO_OUTPUT_FLAG_NONE, &AudioCallback, this, 0
        /*notificationFrames*/);
    mLatencyUs = (int64_t)mAudioTrack->latency() * 1000;
    mFrameSize = mAudioTrack->frameSize();//帧大小
    mAudioTrack->start();
}
mStarted = true;
mPlaying = true;
mPinnedTimeUs = -1ll;
return OK;
}

```

这里要介绍一下 `mAudioSink`。当 `mAudioSink` 不为 `NULL` 的时候，`AudioPlayer` 会将其传入构造函数，而且 `AudioPlayer` 中的播放操作都会依靠 `mAudioSink` 来完成。

此处的 `mAudioSink` 是从 `MediaPlayerService` 注册而来的 `AudioOut` 对象。具体代码在 `MediaPlayerService` 中：


```
status_t MediaPlayerService::Client::setDataSource(*)
{
    //省略部分代码
    mAudioOutput = new AudioOutput();
    static_cast<MediaPlayerInterface*>(p.get())-> setAudioSink(mAudioOutput);
    //省略部分代码
}
```

间接地调用了 StagefrightPlayer->setAudioSink，最终回到 AwesomePlayer 中，代码如下：

```
void AwesomePlayer::setAudioSink(
    const sp<MediaPlayerBase::AudioSink> &audioSink) {
    Mutex::Autolock autoLock(mLock);
    mAudioSink = audioSink;
}
```

构造 AudioPlayer 时用到的就是 mAudioSink 成员变量，因此后面在分析传入的 mAudioSink 操作时，记住实际的对象为 AudioOut 对象，其在 MediaPlayerService 中定义。

4.4.4 AudioPlayer 在 AwesomePlayer 中的运行过程

下面看看 AudioPlayer 的构造函数：

```
AudioPlayer::AudioPlayer(const sp<MediaPlayerBase::AudioSink> &audioSink)
: mAudioTrack(NULL),
  mInputBuffer(NULL),
  mSampleRate(0),
  mLatencyUs(0),
  mFrameSize(0),
  mNumFramesPlayed(0),
  mPositionTimeMediaUs(-1),
  mPositionTimeRealUs(-1),
  mSeeking(false),
  mReachedEOS(false),
  mFinalStatus(OK),
  mStarted(false),
  mIsFirstBuffer(false),
  mFirstBufferResult(OK),
  mFirstBuffer(NULL),
  mAudioSink(audioSink) {
}
```

该构造函数主要用于初始化，并将传入的 audioSink 引用赋值给 mAudioSink 成员变量。再回到前面调用 AudioPlayer 的 start 函数，总结 start 函数内部的过程如下。

- 调用 mSource->read 启动解码，解码第 1 帧相当于启动了解码循环。

- 获取音频参数：采样率、声道数以及量化位数（这里只支持 PCM_16_BIT）。
- 启动输出：这里若 mAudioSink 非空，则启动 mAudioSink 进行输出，否则构造一个 AudioTrack 进行音频输出。这里的 AudioTrack 是比较底层的接口，AudioOut 是 AudioTrack 的封装。

在 start 函数中主要是调用 mAudioSink 进行工作的，主要代码如下：

```
status_t err = mAudioSink->open(mSampleRate, numChannels, channelMask,
AUDIO_FORMAT_PCM_16_BIT,
DEFAULT_AUDIOSINK_BUFFERCOUNT,
&AudioPlayer::AudioSinkCallback,
this,
(mAllowDeepBuffering ?
AUDIO_OUTPUT_FLAG_DEEP_BUFFER :
AUDIO_OUTPUT_FLAG_NONE));
mAudioSink->start();
```

前面介绍过 mAudioSink 是 AudioOut 对象，让我们看看其实现（代码在 mediaplayerservice.cpp 中）。首先分析 mAudioSink 的 open 函数，需要注意的是，传入的参数中有一个函数指针 AudioPlayer::AudioSinkCallback，在 AudioPlayer 中播放 PCM 音频原始数据时会定期调用此回调函数填充到解码后的缓冲队列中，具体实现代码如下：

```
status_t MediaPlayerService::AudioOutput::open(
    uint32_t sampleRate, int channelCount,
    audio_channel_mask_t channelMask,
    audio_format_t format, int bufferCount,
    AudioCallback cb, void *cookie,
    audio_output_flags_t flags) {
    //1
    mCallback = cb;
    mCallbackCookie = cookie;
    //省略部分代码

    int afSampleRate;
    int afFrameCount;
    uint32_t frameCount;
    //省略部分代码
    frameCount = (sampleRate*afFrameCount*bufferCount)/afSampleRate;
    //省略部分代码
    //2
    AudioTrack *t;
    CallbackData *newcbd = NULL;
    if (mCallback != NULL) {
        newcbd = new CallbackData(this);
```

```

        t = new AudioTrack(参数较长省略);
    } else {
        t = new AudioTrack(参数较长省略);
    }

    //3
    mTrack = t;
    status_t res = t->setSampleRate(mPlaybackRatePerMille * mSampleRateHz
/ 1000);
    if (res != NO_ERROR) {
        return res;
    }
    t->setAuxEffectSendLevel(mSendLevel);
    return t->attachAuxEffect(mAuxEffectId);
}

```

总结一下上面的代码：（1）处理传入的参数，回调函数保存在 mCallback 中，cookie 代表的是 AudioPlayer 对象指针类型，接下来根据采样率、声道数等计算 frameCount；（2）构造 AudioTrack 对象，并且赋值给 t；（3）将 AudioTrack 对象存储在 mTrack 成员变量中。

当以上过程完成，继续分析 AudioPlayer 的 start 函数时，最后都会实例化一个 AudioTrack 对象，然后获取音频帧大小、采样率等信息，接着调用 AudioTrack 中的 start 函数，最后到达 MediaPlayerService 服务类，进行音频输出，代码如下：

```

void MediaPlayerService::AudioOutput::start()
{
    ALOGV("start");
    if (mCallbackData != NULL) {
        mCallbackData->endTrackSwitch();
    }
    if (mTrack) {
        mTrack->setVolume(mLeftVolume, mRightVolume);
        mTrack->setAuxEffectSendLevel(mSendLevel);
        mTrack->start();
    }
}

```

在调用 AudioTrack 的 start 函数后，AudioTrack 启动后就会周期性地调用回调函数从解码器获取数据。

4.4.5 音视频同步

回到前面的问题：音频和视频如何同步？答案是通过 fillBuffer，不断填充 Buffer，代码如下：


```

size_t AudioPlayer::fillBuffer(void *data, size_t size) {
    while (size_remaining > 0) {
        MediaSource::ReadOptions options;
        if (mInputBuffer == NULL) {
            status_t err;
            if (mIsFirstBuffer) {
            } else {
                err = mSource->read(&mInputBuffer, &options);
            }
            //省略部分代码
            int64_t timeToCompletionUs = (1000000ll * numFramesPendingPayout)
/ mSampleRate;
            mInputBuffer->meta_data()->findInt64(kKeyTime,
&mPositionTimeMediaUs);
            if (refreshSeekTime) {
                if (useOffload()) {
                    if (postSeekComplete) {
                        ALOGV("fillBuffer is going to post SEEK_COMPLETE");
                        mObserver->postAudioSeekComplete();
                        postSeekComplete = false;
                    }
                    mStartPosUs = mPositionTimeMediaUs;//持有时间戳
                }
                //省略部分代码
            }
        }
        return size_done;
    }
}

```

总结一下上面的代码，当 callback 函数回调 AudioPlayer 读取解码后的数据时，AudioPlayer 会取得两个时间戳 mPositionTimeMediaUs 和 mPositionTimeRealUs。mPositionTimeMediaUs 是数据里面所持有的时间戳（timestamp），mPositionTimeRealUs 则是播放此数据的实际时间（依据帧数目及采样率得出），具体代码如下：

```

int64_t AudioPlayer::getMediaTimeUs() {
    Mutex::Autolock autoLock(mLock);
    //省略部分代码
    if (useOffload()) {
        if (mSeeking) {
            return mSeekTimeUs;
        }
        if (mReachedEOS) {
            int64_t durationUs;
            mSource->getFormat()->findInt64(kKeyDuration, &durationUs);
            return durationUs;
        }
    }
}

```



```

    }
    mPositionTimeRealUs = getOutputPlayPositionUs_1();
    ALOGV("getMediaTimeUs  getOutputPlayPositionUs_1()  mPositionTimeRealUs
%" PRId64, mPositionTimeRealUs);
    return mPositionTimeRealUs;
}
if (mPositionTimeMediaUs < 0 || mPositionTimeRealUs < 0) {
    return mSeekTimeUs;
}

int64_t realTimeOffset = getRealTimeUsLocked() - mPositionTimeRealUs;
if (realTimeOffset < 0) {
    realTimeOffset = 0;
}
return mPositionTimeMediaUs + realTimeOffset;
}

```

总结一下上面的代码：

- 在构造 AudioPlayer 的时候会执行 `mTimeSource = mAudioPlayer`，理解为将 AudioPlayer 作为参考时钟。
- 上述代码中成员变量 `mSeekTimeUs` 是由如下语句获得的：`CHECK(mVideoBuffer -> meta_data() -> findInt64(kKeyTime, &timeUs));`。
- `realTimeOffset = getRealTimeUsLocked() - mPositionTimeRealUs`;表示当显示画面是第 1 帧时，当前音频的播放时间与第 1 帧视频渲染出的时间差值。
- 其中变量 `mPositionTimeRealUs` 是通过在 AudioPlayer 的 `getMediaTimeMapping` 函数中赋值得到的。

跟踪 `getMediaTimeMapping` 函数，你会发现，实际上就是对象 AudioPlayer 里面的 `mPositionTimeRealUs-mPositionTimeMediaUs`。`mPositionTimeRealUs` 实际上是在 `AudioPlayer::fillBuffer` 里面被赋值的，`((mNumFramesPlayed + size_done / mFrameSize) * 1000000) / mSampleRate`;就是目前已经播放完的数据的时间，也就是理想情况下此帧需要被播放的时间。`mPositionTimeMediaUs` 实际上就是 `CHECK(mInputBuffer -> meta_data() -> findInt64(kKeyTime, &mPositionTimeMediaUs))`，即此音频帧在媒体文件中的时间戳。

这样 `mTimeSourceDeltaUs=realTimeUs - mediaTimeUs`;所表示的含义就是该帧音频实际需要播放的时间点和所标注的时间戳的误差，代码如下：

```

bool AudioPlayer::getMediaTimeMapping(
    int64_t *realtime_us, int64_t *mediatime_us) {
    Mutex::Autolock autoLock(mLock);
    if (useOffload()) {
        mPositionTimeRealUs = getOutputPlayPositionUs_1();
    }
}

```

```

        *realtime_us = mPositionTimeRealUs;
        *mediatime_us = mPositionTimeRealUs;
    } else {
        *realtime_us = mPositionTimeRealUs;
        *mediatime_us = mPositionTimeMediaUs;
    }
    return mPositionTimeRealUs != -1 && mPositionTimeMediaUs != -1;
}

```

二者的差值表示这一包 PCM 数据已经播放了多少。StageFright 中的 Video 便依据从 AudioPlayer 得出的两个时间戳的差值作为播放的依据。

4.4.6 音视频输出

最后回到前面的 onVideoEvent 函数中：

```

void AwesomePlayer::onVideoEvent()
{
    //省略部分代码
    mVideoRenderer->render(mVideoBuffer);
    //省略部分代码
}

```

调用 VideoRenderer 的 render 函数，传入视频数据后，就开始通过渲染器渲染出视频，在应用层上可以通过 SurfaceView 或者 TextureView 显示视频画面。

输出过程如图 4-4 所示。

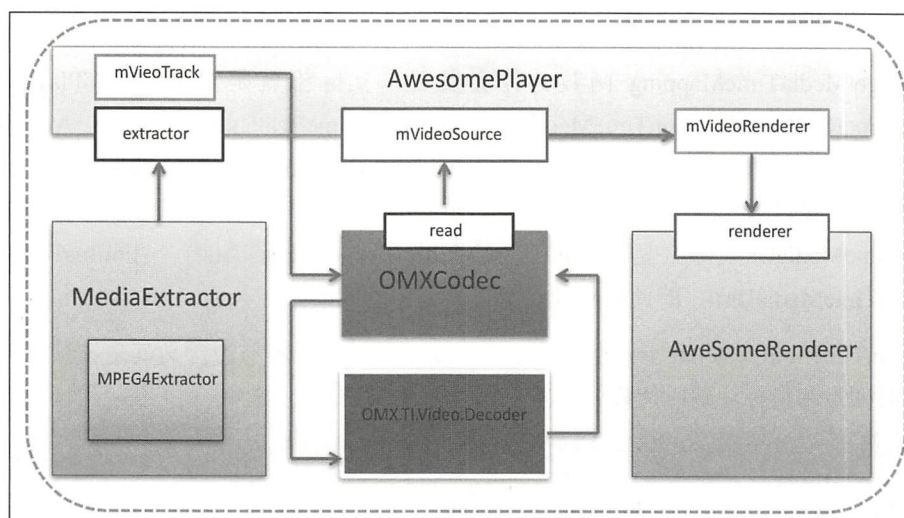


图 4-4 输出过程

从图 4-4 中可以看到，分离出的 `mVideoTrack` 送达 `OMXCodec` 后，调用 `OMX` 组件进行解码，而 `mVideoSource` 不断从 `OMXCodec` 读取解码后的数据，然后发给 `mVideoRenderer` 进行渲染。

4.5 概要总结

本节总结一下整个流程。

可以对照图 4-4，了解到如下要点。

- 设置 `DataSource`，数据源可以分为 `URI` 和 `FD`。`URI` 可以是 `http://`、`rtsp://` 等。`FD` 是一个本地文件描述符，通过 `FD` 可以找到对应的文件。
- 由 `DataSource` 生成 `MediaExtractor`。通过 `sp extractor = MediaExtractor::Create(dataSource)` 来实现。`MediaExtractor::Create(dataSource)` 会根据不同的数据内容创建不同的数据读取对象。
- 通过调用 `setVideoSource`，由 `MediaExtractor` 分解生成音频数据流 (`mAudioTrack`) 和视频数据流 (`mVideoTrack`)。
- 当使用 `onPrepareAsyncEvent` 时，如果 `DataSource` 是 `URL`，根据地址获取数据，并开始缓冲，直到获取 `mVideoTrack` 和 `mAudioTrack`。`mVideoTrack` 和 `mAudioTrack` 通过调用 `initVideoDecoder` 和 `initAudioDecoder` 函数来生成 `mVideoSource` 和 `mAudioSource` 这两个音视频解码器。然后调用 `postBufferingEvent_1` 函数提交事件开启缓冲。
- 数据缓冲的执行函数是 `onBufferingUpdate`。当缓冲区有足够的可以播放数据时，调用 `play_1` 函数开始播放。`play_1` 函数中的关键是调用了 `postVideoEvent_1` 函数，提交了 `mVideoEvent`。这个事件在执行时会调用函数 `onVideoEvent`，这个函数通过调用 `mVideoSource->read(&mVideoBuffer, &options)` 进行视频解码。而音频解码通过 `mAudioPlayer` 实现。
- 视频解码器解码后通过 `mVideoSource->read` 读取一帧帧的数据，将数据放到 `mVideoBuffer` 中，最后通过 `mVideoRenderer->render(mVideoBuffer)` 把视频数据发送到显示模块。当需要暂停或停止时，调用 `cancelPlayerEvents` 函数来提交事件，用于停止解码，还可以选择是否继续缓冲数据。

第 5 章

流媒体播放的新生力量 NuPlayer

Android 2.3 中引入了流媒体框架，而流媒体框架的核心是 NuPlayer 播放器。在之前的版本中，一般认为本地播放就用 StagefrightPlayer (AwesomePlayer)，流媒体就用 NuPlayer。Android 4.0 之后 HTTPLive 和 RTSP 协议开始使用 NuPlayer 播放器，Android 5.0 (L 版本) 之后本地播放也开始使用 NuPlayer 播放器。Android 7.0 (N 版本) 则完全去掉了 AwesomePlayer。由于本书内容基于 Android 6.0 源码，所以前面的章节仍介绍了 AwesomePlayer。

在实现上 NuPlayer 和 AwesomePlayer 不同，NuPlayer 基于 StagefrightPlayer 的基础类构建，利用了底层的 ALooper/AHandler 机制来进行异步解码播放，ALooper 轮循队列中的消息，把消息推送到 AHandler 中处理，最后通过 handleMessage 函数回调，做相应的逻辑处理。

5.1 NuPlayer 整体结构

NuPlayer 是从 MediaPlayerFactory 构造出来的实例 NuPlayerFactory 产生的，其结构关系图如图 5-1 所示。

MediaPlayerFactory 通过工厂模式创建 StagefrightFactory 和 NuPlayerFactory，然后通过 NuPlayerFactory 创建 NuPlayerDriver，接着通过 NuPlayerDriver 构建一个 NuPlayer，NuPlayer 作为播放器，其中涉及数据解析、解码、渲染等过程。

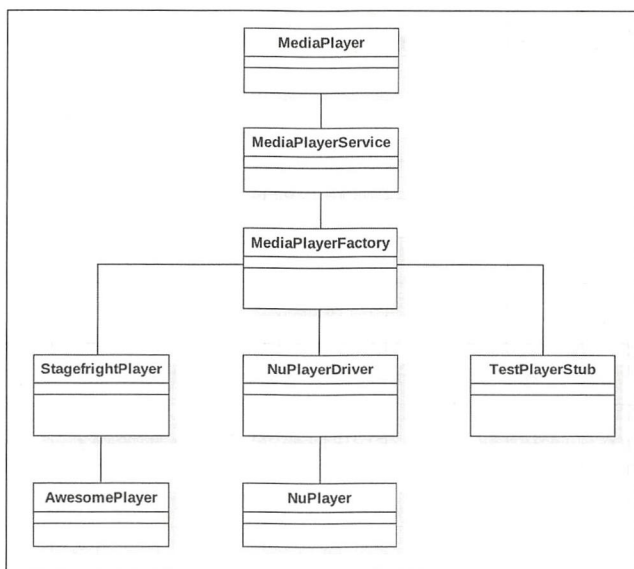


图 5-1 NuPlayer 的结构关系图

NuPlayer 主要用于处理流媒体播放，自然会涉及通过不同流媒体协议传输过来的数据，并有对应的解析和处理逻辑，下面看看 NuPlayer 的类关系图，如图 5-2 所示。

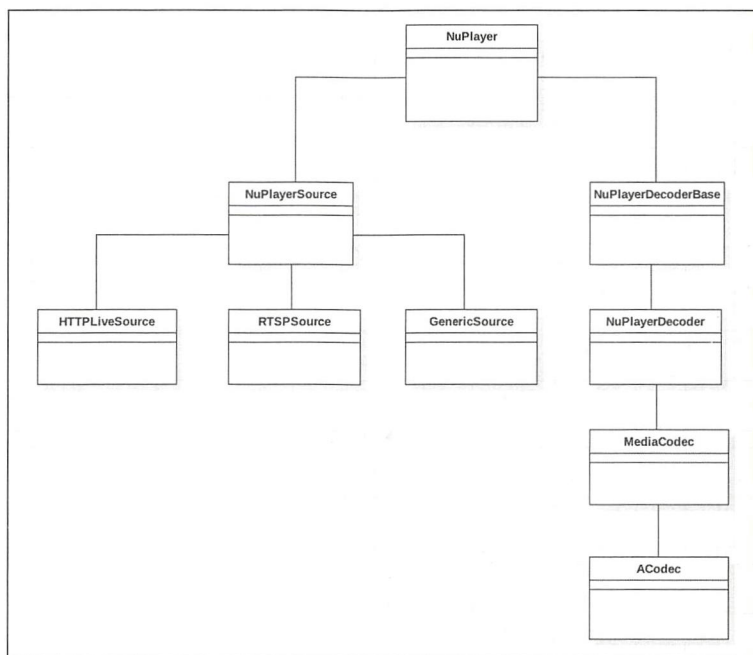


图 5-2 NuPlayer 的类关系图

5.2 NuPlayer 的构建过程

NuPlayer 的构建过程，比较绕。并不像我们想象的那样，在 Java 层创建一个 player 后就构建好一个 NuPlayer，而是在上层调用 setDataSource 函数后，开始到达 MediaPlayerService 中的 setDataSource 函数，通过 getPlayerType 函数获取播放器类型，代码如下：

```
status_t MediaPlayerService::Client::setDataSource(
    const sp<IStreamSource> &source) {
    //创建正确类型的播放器
    player_type playerType = MediaPlayerFactory::getPlayerType(this, source);
    sp<MediaPlayerBase> p = setDataSource_pre(playerType);
    //省略部分代码
    setDataSource_post(p, p->setDataSource(source));
    return mStatus;
}
```

播放器类型枚举如下：

```
enum player_type {
    STAGEFRIGHT_PLAYER = 3,
    NU_PLAYER = 4,
    TEST_PLAYER = 5,
};
```

这时再回到 MediaPlayerFactory 中看看 getPlayerType 函数：

```
player_type MediaPlayerFactory::getPlayerType(const sp<IMediaPlayer>&
client,const sp<IStreamSource> &source) {
    GET_PLAYER_TYPE_IMPL(client, source);
}
```

接下来进入一个宏函数，我们知道宏的主要作用是替换，提高代码的执行效率，毕竟省去了分配和释放栈帧、传参、传返回值等一系列工作。宏函数代码如下：

```
#define GET_PLAYER_TYPE_IMPL(a...) \
    Mutex::Autolock lock_(&sLock); \
    player_type ret = STAGEFRIGHT_PLAYER; \
    float bestScore = 0.0; \
    for (size_t i = 0; i < sFactoryMap.size(); ++i) { \
        IFactory* v = sFactoryMap.valueAt(i); \
        float thisScore; \
        CHECK(v != NULL); \
        thisScore = v->scoreFactory(a, bestScore); \
        if (thisScore > bestScore) { \
            ret = sFactoryMap.keyAt(i); \
            bestScore = thisScore; \
        } \
    }
```



```

    }
}
if (0.0 == bestScore) {
    ret = getDefaultPlayerType();
}
return ret;

```

这个宏函数的标示遍历 map 中存放的播放器工厂类，调用 scoreFactory 可以得到播放器的播放能力。这时候根据前面 StagefrightPlayerFactory 中的 if 判断逻辑，thisScore > bestScore 条件不成立，所以得到 thisScore 是 0.0，而如果是 NuPlayer，默认就会有一个 0.8 的值，所以返回的 ret 就是 NuPlayerFactory 对象。如果得到的值是 0.0，就会进入 getDefaultPlayerType 函数，代码如下：

```

static player_type getDefaultPlayerType() {
    char value[PROPERTY_VALUE_MAX];
    if (property_get("media.stagefright.use-awesome", value, NULL)
        && (!strcmp("1", value) || !strcasecmp("true", value))) {
        return STAGEFRIGHT_PLAYER;
    }
    return NU_PLAYER;
}

```

也就是如果设置了 property 是 media.stagefright.use-awesome，才会走到 StagefrightPlayerFactory，默认是 NuPlayerFactory。可见 Google 已经逐步替换掉 StagefrightPlayer 而使用 NuPlayer 了。

针对 StagefrightPlayerFactory 会创建 StagefrightPlayer，而针对 NuPlayerFactory 不会直接创建 NuPlayer，而是在 NuPlayerDriver 的构造函数中创建一个 NuPlayerDriver：

```

NuPlayerDriver::NuPlayerDriver(pid_t pid)
: mState(STATE_IDLE),
  mIsAsyncPrepare(false),
  mAsyncResult(UNKNOWN_ERROR),
  mSetSurfaceInProgress(false),
  mDurationUs(-1),
  mPositionUs(-1),
  mSeekInProgress(false),
  mLooper(new ALooper),
  mPlayerFlags(0),
  mAtEOS(false),
  mLooping(false),
  mAutoLoop(false),
  mStartupSeekTimeUs(-1) {
    ALOGV("NuPlayerDriver(%p)", this);
}

```

```

mLooper->setName("NuPlayerDriver Looper");
mLooper->start(
    false, /* runOnCallingThread */
    true,  /* canCallJava */
    PRIORITY_AUDIO);
mPlayer = new NuPlayer(pid);
mLooper->registerHandler(mPlayer);
mPlayer->setDriver(this);
}

```

在 NuPlayerDriver 的构造函数中，创建了一个 NuPlayer，NuPlayerDriver 是对 NuPlayer 的封装，继承 MediaPlayerInterface 接口。通过 NuPlayer 可以实现播放功能，主要用于通知上面的流程。

NuPlayer 继承自 AHandler，并且引入了 AMessage，通过 ALooper 来处理消息，如 NuPlayerDriver 调用 NuPlayer 的 prepareAsync 函数：

```

void NuPlayer::prepareAsync() {
    (new AMessage(kWhatPrepare, this))->post();
}

```

实际就是发送一个 kWhatPrepare 消息。在 onMessageReceived 函数中，收到消息并进行处理：

```

void NuPlayer::onMessageReceived(const sp<AMessage> &msg) {
    switch (msg->what()) {
        //省略部分代码
        case kWhatPrepare:
        {
            mSource->prepareAsync();
            break;
        }
        //省略部分代码
    }
}

```

5.3 NuPlayer 的数据解析模块

从 NuPlayer 的框架结构看，解析的模块主要是 NuPlayerSource 以及继承它的几个类，如 HTTPLiveSource、RTSPSource、GenericSource 等。

在 NuPlayer 中调用 setDataSourceAsync 函数后：

```

void NuPlayer::setDataSourceAsync(
    const sp<IMediaHTTPService> &httpService,

```



```

        const char *url,
        const KeyedVector<String8, String8> *headers) {
    sp<AMessage> msg = new AMessage(kWhatSetDataSource, this);
    size_t len = strlen(url);
    sp<AMessage> notify = new AMessage(kWhatSourceNotify, this);
    sp<Source> source;
    if (IsHTTPLiveURL(url)) { //通过 HTTP 协议传输数据
        source = new HTTPLiveSource(notify, httpService, url, headers);
    } else if (!strncasecmp(url, "rtsp://", 7)) { //通过 RTSP 协议传输数据
        source = new RTSPSource(
            notify, httpService, url, headers, mUIDValid, mUID);
    } else if ((!strncasecmp(url, "http://", 7)
        || !strncasecmp(url, "https://", 8))
        && ((len >= 4 && !strncasecmp(".sdp", &url[len - 4]))
        || strstr(url, ".sdp?"))) { //通过 SDP 协议的网络数据
        source = new RTSPSource(
            notify, httpService, url, headers, mUIDValid, mUID, true);
    } else { //本地媒体文件
        sp<GenericSource> genericSource =
            new GenericSource(notify, mUIDValid, mUID);
        status_t err = genericSource->setDataSource(httpService, url,
headers);
    }
    msg->setObject("source", source);
    msg->post();
}

```

这里会根据不同协议选择不同的 Source（视频源）对象，有了这个 Source（视频源）对象后，发送 kWhatSetDataSource 消息，代码如下：

```

case kWhatSetDataSource:
{
    sp<RefBase> obj;
    CHECK(msg->findObject("source", &obj));
    if (obj != NULL) {
        mSource = static_cast<Source*>(obj.get());
        //强制转换成 mSource, 给 Decoder*（解码器）使用
    } else {
        err = UNKNOWN_ERROR;
    }
    CHECK(mDriver != NULL);
    sp<NuPlayerDriver> driver = mDriver.promote();
    if (driver != NULL) {
        driver->notifySetDataSourceCompleted(err);
    }
}

```



```
break;
}
```

也就是通过具体的 Source 解析完数据，再把 Source 强制转换成 mSource 给 Decoder 使用，这时里面就包含了数据相关信息，如几个 Track、是什么格式的等。其中的一个 HTTPLiveSource，主要用于解析 HLS 协议。图 5-3 是 HTTPLiveSource 的结构图。

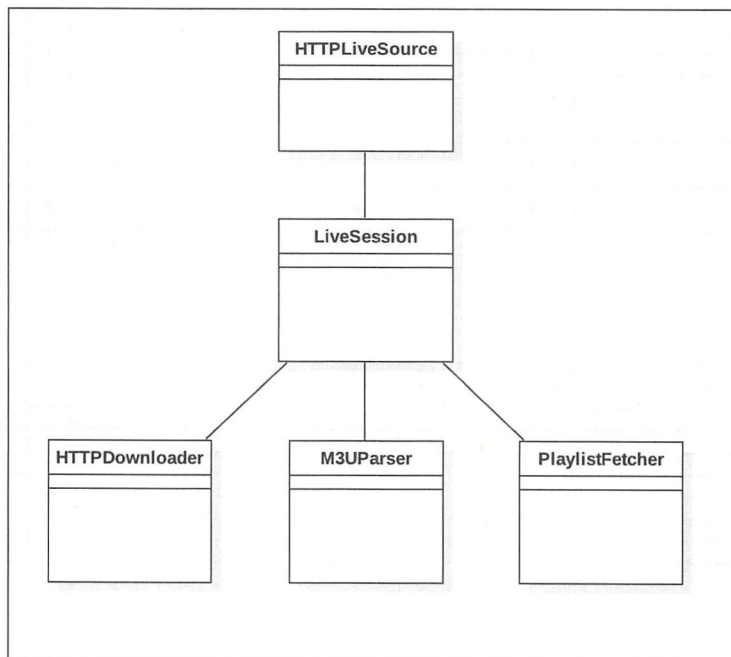


图 5-3 HTTPLiveSource 的结构图

图 5-3 里面主要封装了 LiveSession。LiveSession 有 HTTPDownloader 模块、M3UParser 模块，以及取播放列表的 PlaylistFetcher 模块。

当调用 HTTPLiveSource 的 prepareAsync 函数时，代码如下：

```
void NuPlayer::HTTPLiveSource::prepareAsync() {
    if (mLiveLooper == NULL) {
        mLiveLooper = new ALooper;
        mLiveLooper->setName("http live");
        mLiveLooper->start();
        mLiveLooper->registerHandler(this);
    }
    sp<AMessage> notify = new AMessage(kWhatSessionNotify, this);
    mLiveSession = new LiveSession(
        notify,
```

```

        (mFlags & kFlagIncognito) ? LiveSession::kFlagIncognito : 0,
        mHTTPService);
    mLiveLooper->registerHandler(mLiveSession);
    mLiveSession->connectAsync(
        mURL.c_str(), mExtraHeaders.isEmpty() ? NULL : &mExtraHeaders);
}

```

内部主要构建了一个 LiveSession 对象。通过 LiveSession 内部的 connectAsync 函数，创建一个会话：

```

void LiveSession::connectAsync(const char *url, const KeyedVector <String8,
String8> *headers) {
    sp<AMessage> msg = new AMessage(kWhatConnect, this);
    msg->setString("url", url);
    msg->post(); //发送 kWhatConnect 消息
}
//接收消息后开始处理
case kWhatConnect:
{
    onConnect(msg);
    break;
}
}

```

接着看看 onConnect 函数：

```

void LiveSession::onConnect(const sp<AMessage> &msg) {
    //省略部分代码 if (!audio)
    //创建一个轮循器
    if (mFetcherLooper == NULL) {
        mFetcherLooper = new ALooper();
        mFetcherLooper->setName("Fetcher");
        mFetcherLooper->start(false, false);
    }
    //创建获取器取播放列表
    addFetcher(mMasterURL.c_str())->fetchPlaylistAsync();
}

```

网络请求，开始取播放列表。在取到播放列表后，会进行回调：

```

void LiveSession::onMasterPlaylistFetched(const sp<AMessage> &msg) {
    size_t initialBandwidth = 0;
    size_t initialBandwidthIndex = 0;
    int32_t maxWidth = 0;
    int32_t maxHeight = 0;
    //省略部分代码
    if (mPlaylist->isVariantPlaylist()) { //是否是有效的播放列表
        Vector<BandwidthItem> itemsWithVideo;
    }
}

```

```

        for (size_t i = 0; i < mPlaylist->size(); ++i) {
            //mPlaylist 是 sp<M3UParser>
            BandwidthItem item;
            item.mPlaylistIndex = i;
            item.mLastFailureUs = -111;
            sp<AMessage> meta;
            AString uri;
            mPlaylist->itemAt(i, &uri, &meta);
            CHECK(meta->findInt32("bandwidth", (int32_t *)&item.mBandwidth));
            int32_t width, height;
            if (meta->findInt32("width", &width)) {
                maxWidth = max(maxWidth, width); //宽度
            }
            if (meta->findInt32("height", &height)) {
                maxHeight = max(maxHeight, height); //高度
            }
            mBandwidthItems.push(item); //存到容器中
            if (mPlaylist->hasType(i, "video")) {
                itemsWithVideo.push(item);
            }
        }
        initialBandwidth = mBandwidthItems[0].mBandwidth;
        mBandwidthItems.sort(SortByBandwidth);
        for (size_t i = 0; i < mBandwidthItems.size(); ++i) {
            if (mBandwidthItems.itemAt(i).mBandwidth == initialBandwidth) {
                initialBandwidthIndex = i;
                break;
            }
        }
    }

    //省略部分代码
    mMaxWidth = maxWidth > 0 ? maxWidth : mMaxWidth;
    mMaxHeight = maxHeight > 0 ? maxHeight : mMaxHeight;
    mPlaylist->pickRandomMediaItems();
    changeConfiguration(011 /* timeUs */, initialBandwidthIndex, false
/* pickTrack */);
}

```

上面的代码主要是根据 URL 返回的 M3U 文件，获取对应的 BandwidthItem，如果熟悉 M3U 文件，可以知道 M3U 文件有一级索引和二级索引，M3U 一级索引内容举例如下：

```

#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=200000
gear1/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=311111

```



```
gear2/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=484444
gear3/prog_index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=737777
gear4/prog_index.m3u8
```

其主要作用是动态码率适应, BANDWIDTH 越大, 分辨率越高。BANDWIDTH 实际上就是带宽。本书后面在流媒体协议相关章节中会详细介绍这部分细节。

例如, 想要获得某个 Track 的信息, 得到视频、音频、字幕相关信息, 可以通过 LiveSession 的 getTrackInfo 函数:

```
sp<AMessage> LiveSession::getTrackInfo(size_t trackIndex) const {
    if (mPlaylist == NULL) {
        return NULL;
    } else {
        if (trackIndex == mPlaylist->getTrackCount() && mHasMetadata) {
            sp<AMessage> format = new AMessage();
            format->setInt32("type", MEDIA_TRACK_TYPE_METADATA);
            format->setString("language", "und");
            format->setString("mime", MEDIA_MIMETYPE_DATA_TIMED_ID3);
            return format;
        }
        return mPlaylist->getTrackInfo(trackIndex);
    }
}
```

到此, 数据解析模块就完成了, 还有一些 Source (视频源) 类型读者可以自行查看代码, 主要作用都是得到音视频数据信息, 最终通过 mSource 变量传递给解码器。

5.4 NuPlayer 的解码模块

NuPlayer 的解码引入了 NuPlayerDecoderBase, 这是一个基类。真正的解码器逻辑在 NuPlayerDecoder.cpp 文件中, 它继承自 NuPlayerDecoderBase 类。下面先看看实例化解码器, 其位于 frameworks\av\media\libmediaplayerservice\nuplayer\NuPlayer.cpp 中:

```
status_t NuPlayer::instantiateDecoder(bool audio, sp<DecoderBase> *decoder) {
    //省略部分代码
    if (!audio) {
        AString mime;
        CHECK(format->findString("mime", &mime));
        sp<AMessage> ccNotify = new AMessage(kWhatClosedCaptionNotify,
this);
```

```

        if (mCCDecoder == NULL) {
            mCCDecoder = new CCDecoder(ccNotify);
        }
    }
    if (audio) { //是音频
        sp<AMessage> notify = new AMessage(kWhatAudioNotify, this);
        if (mOffloadAudio) {
            const bool hasVideo = (mSource->getFormat(false /*audio */) !=
NULL);

            format->setInt32("has-video", hasVideo);
            *decoder = new DecoderPassThrough(notify, mSource, mRenderer);
        } else {
            *decoder = new Decoder(notify, mSource, mPID, mRenderer);
        }
    } else { //是视频
        sp<AMessage> notify = new AMessage(kWhatVideoNotify, this);
        *decoder = new Decoder(
            notify, mSource, mPID, mRenderer, mSurface, mCCDecoder);
    }
    (*decoder)->init(); //解码器初始化
    (*decoder)->configure(format); //解码器构建
    //分配缓冲区给原生的视频数据
    if (!audio && (mSourceFlags & Source::FLAG_SECURE)) {
        Vector<sp<ABuffer> > inputBufs;
        Vector<MediaBuffer *> mediaBufs;
        for (size_t i = 0; i < inputBufs.size(); i++) {
            const sp<ABuffer> &buffer = inputBufs[i];
            MediaBuffer *mbuf = new MediaBuffer(buffer->data(), buffer->
size());
            mediaBufs.push(mbuf);
        }
        status_t err = mSource->setBuffers(audio, mediaBufs);
        //省略部分代码
    }
    return OK;
}

```

上面的代码构造了 Decoder 对象，且进行了初始化和构建解码器。

由于 NuPlayerDecoder 继承自 NuPlayerDecoderBase 类，所以 Configure 函数会执行 NuPlayerDecoder 的 Configure 函数，最终回调 onConfigure 函数，其位于 frameworks\av\media\libmediaplayerservice\nuplayer\NuPlayerDecoder.cpp 中，代码如下：

```

void NuPlayer::Decoder::onConfigure(const sp<AMessage> &format) {
    //省略部分代码
    AString mime;

```



```

CHECK(format->findString("mime", &mime));
mIsAudio = !strncasecmp("audio/", mime.c_str(), 6); //mime 是否是音频
mIsVideoAVC = !strcasecmp(MEDIA_MIMETYPE_VIDEO_AVC, mime.c_str());
//mime 是否是 H.264 编码格式的视频
mComponentName = mime;
mComponentName.append(" decoder");
ALOGV("[%s] onConfigure (surface=%p)", mComponentName.c_str(),
mSurface.get());
//通过类型创建编解码器
mCodec = MediaCodec::CreateByType(mCodecLooper, mime.c_str(), false
/* encoder */, NULL /* err */, mPid);
int32_t secure = 0;
if (format->findInt32("secure", &secure) && secure != 0) {
    if (mCodec != NULL) {
        mCodec->getName(&mComponentName);
        mComponentName.append(".secure");
        mCodec->release();
        ALOGI("[%s] creating", mComponentName.c_str());
        //通过组件名创建编解码器
        mCodec = MediaCodec::CreateByComponentName(
            mCodecLooper, mComponentName.c_str(), NULL /* err */,
mPid);
    }
}
mCodec->getName(&mComponentName);
//进行构建
err = mCodec->configure(
    format, mSurface, NULL /* crypto */, 0 /* flags */);
mStats->setString("mime", mime.c_str());
mStats->setString("component-name", mComponentName.c_str());
sp<AMessage> reply = new AMessage(kWhatCodecNotify, this);
mCodec->setCallback(reply);
err = mCodec->start();
//省略部分代码
}

```

从 onConfigure 函数可以看到, NuPlayerDecoder 引入了 MediaCodec 作为解码器。通过 CreateByType/CreateByComponentName 创建了 Codec 对象。由于 MediaCodec 后面会专门介绍, 这里就不做过多介绍了。

5.5 NuPlayer 的渲染模块

渲染模块的主要功能如下。

Android 音视频开发

- 将音视频原始数据缓存到队列。
- 音频数据消耗播放。
- 视频数据消耗显示。
- 音视频同步。
- 播放器控制。

下面将音视频原始数据缓存到队列。在 `\frameworks\av\media\libmediaplayerservice\nuplayer\NuPlayerRenderer.cpp` 中, 存在一个 `QueueEntry` 结构体和两个队列, 代码如下:

```
struct QueueEntry {
    sp<ABuffer> mBuffer;
    sp<AMessage> mNotifyConsumed;
    size_t mOffset;
    status_t mFinalResult;
    int32_t mBufferOrdinal;
};
List<QueueEntry> mAudioQueue; //音频缓存队列
List<QueueEntry> mVideoQueue; //视频缓存队列
```

那么数据是如何添加的呢?在 `onQueueBuffer` 函数中进行添加:

```
void NuPlayer::Renderer::onQueueBuffer(const sp<AMessage> &msg) {
    int32_t audio;
    CHECK(msg->findInt32("audio", &audio));
    if (dropBufferIfStale(audio, msg)) {
        return;
    }

    if (audio) {
        mHasAudio = true; //音频
    } else {
        mHasVideo = true; //视频
    }
    if (mHasVideo) {
        if (mVideoScheduler == NULL) {
            mVideoScheduler = new VideoFrameScheduler(); //做视频渲染调整
            mVideoScheduler->init();
        }
    }
    //省略部分代码
    QueueEntry entry;
    entry.mBuffer = buffer;
    entry.mNotifyConsumed = notifyConsumed;
    entry.mOffset = 0;
```

```

entry.mFinalResult = OK;
entry.mBufferOrdinal = ++mTotalBuffersQueued;
if (audio) {
    Mutex::Autolock autoLock(mLock);
    mAudioQueue.push_back(entry);
    //将音频数据放入音频队列, 注意这里是一个结构体, 可以理解为一个 packet
    postDrainAudioQueue_1(); //刷新音频
} else {
    mVideoQueue.push_back(entry);
    //将视频数据放入视频队列, 注意这里是一个结构体, 可以理解为一个 packet
    postDrainVideoQueue(); //刷新视频
}

Mutex::Autolock autoLock(mLock);
if (!mSyncQueues || mAudioQueue.empty() || mVideoQueue.empty()) {
    return;
}
sp<ABuffer> firstAudioBuffer = (*mAudioQueue.begin()).mBuffer;
sp<ABuffer> firstVideoBuffer = (*mVideoQueue.begin()).mBuffer;
if (firstAudioBuffer == NULL || firstVideoBuffer == NULL) {
    //在队列中, 发出 EOS 信号
    syncQueuesDone_1();
    return;
}

int64_t firstAudioTimeUs;
int64_t firstVideoTimeUs;
int64_t diff = firstVideoTimeUs - firstAudioTimeUs;
if (diff > 10000011) {
    //音频比视频长 0.1s, 丢弃音频数据
    (*mAudioQueue.begin()).mNotifyConsumed->post();
    mAudioQueue.erase(mAudioQueue.begin());
    return;
}
syncQueuesDone_1();

```

那么音频是怎么播放的呢? 在 NuPlayerRenderer 中, 会先调用 openAudioSink 函数:

```

status_t NuPlayer::Renderer::openAudioSink(
    const sp<AMessage> &format,
    bool offloadOnly,
    bool hasVideo,
    uint32_t flags,
    bool *isOffloaded) {
    sp<AMessage> msg = new AMessage(kWhatOpenAudioSink, this);

```


Android 音视频开发

```
msg->setMessage("format", format);
msg->setInt32("offload-only", offloadOnly);
msg->setInt32("has-video", hasVideo);
msg->setInt32("flags", flags);
sp<AMessage> response;
msg->postAndAwaitResponse(&response);
//省略部分代码
return err;
```

发送一个 `kWhatOpenAudioSink` 消息。收到消息后进行如下处理：

```
case kWhatOpenAudioSink:
{
    sp<AMessage> format;
    status_t err = onOpenAudioSink(format, offloadOnly, hasVideo, flags);
    //省略部分代码
    break;
}
```

调用 `onOpenAudioSink` 函数，代码如下：

```
status_t NuPlayer::Renderer::onOpenAudioSink(
    const sp<AMessage> &format, bool offloadOnly, bool hasVideo,
    uint32_t flags) {
    //省略部分代码
    status_t err = mAudioSink->open(sampleRate, numChannels, (audio_
channel_mask_t) channelMask, AUDIO_FORMAT_PCM_16_BIT, 0 /* bufferCount - unused
*/, mUseAudioCallback ? &NuPlayer::Renderer::AudioSinkCallback : NULL,
mUseAudioCallback ? this : NULL, (audio_output_flags_t)pcmFlags, NULL,
doNotReconnect, frameCount);
    if (err == OK) {
        err = mAudioSink->setPlaybackRate(mPlaybackSettings);
    }
    mCurrentPcmInfo = info;
    if (!mPaused) { //在预览模式下，如果是暂停状态，不要播放
        mAudioSink->start();
    }
}
if (audioSinkChanged) {
    onAudioSinkChanged();
}
mAudioTornDown = false;
return OK;
}
```

这里打开了 `AudioSink`（音频后端）。当将音频数据放入音频队列中时，会接着调用

postDrainAudioQueue_1 函数:

```
void NuPlayer::Renderer::postDrainAudioQueue_1(int64_t delayUs) {
    //省略部分代码
    mDrainAudioQueuePending = true;
    sp<AMessage> msg = new AMessage(kWhatDrainAudioQueue, this);
    msg->setInt32("drainGeneration", mAudioDrainGeneration);
    msg->post(delayUs);
}
```

上面主要是发送了一个消息 kWhatDrainAudioQueue, 找到对应接收消息的地方, 代码如下:

```
case kWhatDrainAudioQueue:
{
    mDrainAudioQueuePending = false;
    if (onDrainAudioQueue()) {
        uint32_t numFramesPendingPayout = mNumFramesWritten -
numFramesPlayed;
        //AudioSink 中缓存的可用于播放的数据时长
        int64_t delayUs = mAudioSink->msecsPerFrame() *
numFramesPendingPayout * 1000ll;
        if (mPlaybackRate > 1.0f) {
            delayUs /= mPlaybackRate;
        }
        Mutex::Autolock autoLock(mLock);
        //利用一半的延时刷新下次数据
        postDrainAudioQueue_1(delayUs / 2);
    }
    break;
}
```

主要是有一个进行判断的 onDrainAudioQueue 函数, 判断是否需要重新向 AudioSink 写入数据, 代码如下:

```
bool NuPlayer::Renderer::onDrainAudioQueue() {
    //当消耗音频数据, 发现队列中的 Buffer 数据无效时, 直接返回 false
    if (mAudioTornDown) {
        return false;
    }
    //省略部分代码
    uint32_t prevFramesWritten = mNumFramesWritten;
    while (!mAudioQueue.empty()) { //只要音频队列不为空
        QueueEntry *entry = &*mAudioQueue.begin();
        mLastAudioBufferDrained = entry->mBufferOrdinal;
        //省略部分代码
    }
```

Android 音视频开发

```

        size_t copy = entry->mBuffer->size() - entry->mOffset;
        //源源不断地向 AudioSink (音频后端) 写入数据, 此时 AudioSink (音频后端) 已
        //经处于 open 状态了, 其可以用于播放音频
        ssize_t written = mAudioSink->write(entry->mBuffer->data() +
entry->mOffset, copy, false /* blocking */);
        //省略部分代码
        entry->mOffset += written;
        if (entry->mOffset == entry->mBuffer->size()) {
            entry->mNotifyConsumed->post(); //通知解码器音频数据已经消耗
            mAudioQueue.erase(mAudioQueue.begin());
            entry = NULL;
        }
        size_t copiedFrames = written / mAudioSink->frameSize();
        mNumFramesWritten += copiedFrames;
        {
            Mutex::Autolock autoLock(mLock);
            int64_t maxTimeMedia;
            maxTimeMedia = mAnchorTimeMediaUs + (int64_t)(max((long
long)mNumFramesWritten - mAnchorNumFramesWritten, 0LL) * 1000LL *
mAudioSink->msecsPerFrame());
            mMediaClock->updateMaxTimeMedia(maxTimeMedia);
            notifyIfMediaRenderingStarted_1();
        }
        break;
    }
}
//判断是否需要重新写入数据
bool reschedule = !mAudioQueue.empty()
    && (!mPaused || prevFramesWritten != mNumFramesWritten);
return reschedule;
}

```

到这里, 已经很清楚, 音频播放流程如下: 先打开音频后端, 然后当向音频队列中发送数据时, 音频队列同时向音频后端写入数据, 以供播放音频。

那么视频显示呢? 同样是在视频原始数据进入视频队列后, 开始执行 `postDrainVideoQueue` 函数:

```

void NuPlayer::Renderer::postDrainVideoQueue() {
    QueueEntry &entry = *mVideoQueue.begin();
    sp<AMessage> msg = new AMessage(kWhatDrainVideoQueue, this);
    msg->setInt32("drainGeneration", getDrainGeneration(false /* audio */));
    //省略部分代码
    int64_t delayUs;
}

```



```

int64_t nowUs = ALooper::GetNowUs();
int64_t realTimeUs;
if (mFlags & FLAG_REAL_TIME) {
    int64_t mediaTimeUs;
    CHECK(entry.mBuffer->meta()->findInt64("timeUs", &mediaTimeUs));
    realTimeUs = mediaTimeUs;
} else {
    int64_t mediaTimeUs;
    {
        Mutex::Autolock autoLock(mLock);
        if (mAnchorTimeMediaUs < 0) { //若时间基 (TimeBase) 未设置, 直接显示
            mMediaClock->updateAnchor(mediaTimeUs, nowUs, mediaTimeUs);
            mAnchorTimeMediaUs = mediaTimeUs;
            realTimeUs = nowUs;
        } else { //其他条件, 以视频为准
            realTimeUs = getRealTimeUs(mediaTimeUs, nowUs);
        }
    }
    if (!mHasAudio) {
        //平滑视频帧率>10fps
        mMediaClock->updateMaxTimeMedia(mediaTimeUs + 100000);
    }
    delayUs = realTimeUs - nowUs;
    if (delayUs > 500000) {
        int64_t postDelayUs = 500000;
        if (mHasAudio && (mLastAudioBufferDrained - entry.mBufferOrdinal)
            <= 0) {
            postDelayUs = 10000;
        }
        msg->setWhat(kWhatPostDrainVideoQueue);
        msg->post(postDelayUs);
        mVideoScheduler->restart();
        ALOGI("possible video time jump of %dms, retrying in %dms",
            (int)(delayUs / 1000), (int)(postDelayUs / 1000));
        mDrainVideoQueuePending = true;
        return;
    }
}
realTimeUs = mVideoScheduler->schedule(realTimeUs * 1000) / 1000;
int64_t twoVsyncsUs = 2 * (mVideoScheduler->getVsyncPeriod() / 1000);
delayUs = realTimeUs - nowUs;
//计算延时, 下一帧什么时候显示
ALOGW_IF(delayUs > 500000, "unusually high delayUs: %" PRId64, delayUs);
//在渲染之前, 延迟 2 个显示刷新

```


Android 音视频开发

```
msg->post(delayUs > twoVsyncsUs ? delayUs - twoVsyncsUs : 0);
mDrainVideoQueuePending = true;
}
```

这里先发送一个 `kWhatDrainVideoQueue` 的消息，然后进行音视频同步。发送 `kWhatDrainVideoQueue` 消息后，会触发调用 `onDrainVideoQueue` 函数：

```
void NuPlayer::Renderer::onDrainVideoQueue() {
    QueueEntry *entry = &mVideoQueue.begin();
    //省略部分代码
    int64_t nowUs = -1;
    int64_t realTimeUs;
    //省略部分代码
    bool tooLate = false;
    if (!mPaused) {
        if (nowUs == -1) {
            nowUs = ALooper::GetNowUs();
        }
        setVideoLateByUs(nowUs - realTimeUs);
        tooLate = (mVideoLateByUs > 40000);
        if (tooLate) {
            ALOGV("video late by %lld us (%.2f secs)",
                (long long)mVideoLateByUs, mVideoLateByUs / 1E6);
        } else {
            int64_t mediaUs = 0;
            mMediaClock->getMediaTime(realTimeUs, &mediaUs);
            ALOGV("rendering video at media time %.2f secs",
                (mFlags & FLAG_REAL_TIME ? realTimeUs :
                    mediaUs) / 1E6);
        }
    } else {
        setVideoLateByUs(0);
        if (!mVideoSampleReceived && !mHasAudio) {
            Mutex::Autolock autoLock(mLock);
            clearAnchorTime_l();
        }
    }
    entry->mNotifyConsumed->setInt64("timestampNs", realTimeUs * 1000ll);
    entry->mNotifyConsumed->setInt32("render", !tooLate);
    entry->mNotifyConsumed->post(); //向解码器发送消息，表示已经消耗数据
    mVideoQueue.erase(mVideoQueue.begin());
    entry = NULL;
    mVideoSampleReceived = true;
    if (!mPaused) {
        if (!mVideoRenderingStarted) {
```

```
        mVideoRenderingStarted = true;
        notifyVideoRenderingStart();
    }
    Mutex::Autolock autoLock(mLock);
    notifyIfMediaRenderingStarted_1(); // 向上通知开始渲染视频
}
}
```

NuPlayer::Renderer 使用的是以视频为基准的同步机制，解码后的音频数据时间戳如果大于视频数据时间戳，直接丢弃音频包，然后直接渲染视频。同步机制主要位于视频缓冲区处理部分的 onDrainVideoQueue 和音频缓冲区处理部分的 onDrainAudioQueue 中。音视频的渲染都采用类似定时器的机制，只不过视频显示需要依赖于实际解码器，音频播放需要依赖于 AudioSink 的接口。



第 6 章

OpenMAX (OMX) 框架

6.1 Codec 部分中的 AwesomePlayer 到 OMX 服务

前两章介绍了 AwesomePlayer 和 NuPlayer，而最终解码都会到达 OMX 框架，也就是 OpenMAX 框架，本节开始分析编解码部分中的 AwesomePlayer 到 OMX 服务过程，也就是开启 OpenMAX 准备相关内容。Android 系统中用 OpenMAX 来做编解码，Android 向上抽象了一层 OMXCodec，提供给上层播放器 AwesomePlayer 使用。同时有一个 IOMX 接口，在 ACodec 中可以通过 IOMX 调用 OpenMAX 组件。播放器中音视频解码器 mVideoSource、mAudioSource 都是 OMXCodec 的实例。

OMXCodec::Create 是解码器初始化的入口。OMXCodec 通过 IOMX 依赖 Binder 机制获得 OMX 服务，OMX 服务才是 OpenMAX 在 Android 中的实现。

6.1.1 OpenMAX 与 StageFright 框架层级的关系

StageFright 框架通过 OpenMAX 与硬件层进行通信，图 6-1 是 OpenMAX 和 StageFright 的层级关系图。

在图 6-1 中可以看到，StrageFright 层共有两路到达 OpenMAX 框架。一路是通过 NuPlayer 到达 ACodec 类，然后直接调用 OMX IL Core 中的接口。另一路是通过 StagefrightPlayer 到 AwesomePlayer，再到达 OMXCodec 类，然后调用 OMX 组件接口进行数据传输。

在以前的 AwesomePlayer 中，音频和视频数据会到 OMXCodec 中寻找对应的解码器进行解码，如图 6-2 所示。IOMX 和 OMX 组件通过 Binder 通信，中间还涉及 OMXClient。OMX



中的 OMXNodeInstance 负责创建并维护不同的实例，这些实例是根据上面的需求创建的，以 Node 作为唯一标识。这样播放器中的每一个 OMXCodec 在 OMX 服务器端都有了自己对应的 OMXNodeInstance 实例。OMXMaster 维护底层软硬件解码库，根据 OMXNodeInstance 中想要的解码器来创建解码实体组件。

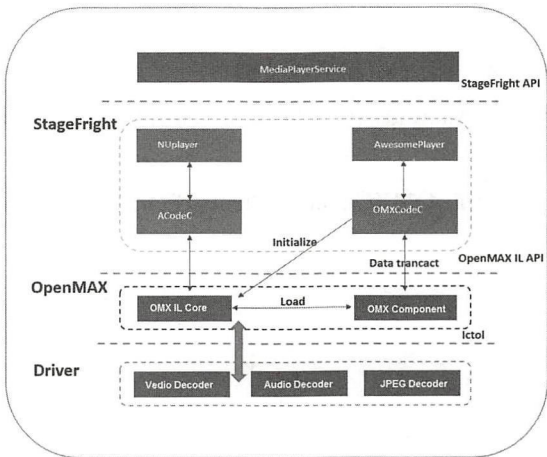


图 6-1 OpenMAX 和 StageFright 的层级关系图

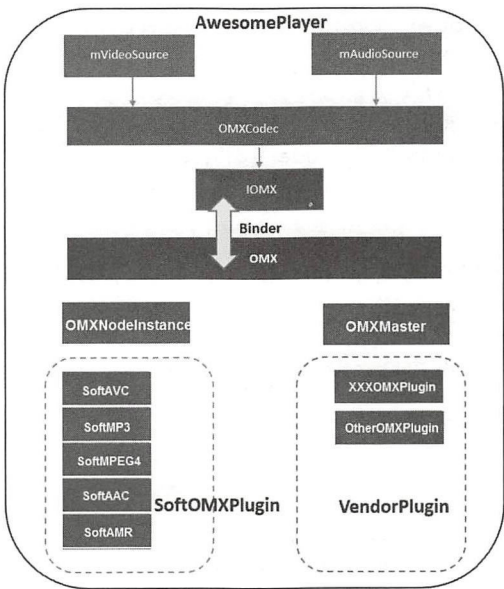


图 6-2 AwesomePlayer 与 OMX 的关系



6.1.2 OMX 的初始化流程

AwesomePlayer 是如何获得 OMX 服务的？过程如下。

- 在 AwesomePlayer 初始化的时候，会调用 AwesomePlayer::onPrepareAsyncEvent。
- 继而调用 AwesomePlayer::initVideoDecoder 以及 AwesomePlayer::initAudioDecoder。
- 然后开始正式进入 OMX 以及硬件解码器的初始化工作。

之前的 AwesomePlayer 初始化工作都是在做铺垫。当 OMX 开始初始化时，才真正开始核心的初始化工作。我们知道，Android 中的组件都是在提供服务，有服务器端，有客户端，大部分是 C/S 模型，在前面的章节中也已经介绍过。在 AwesomePlayer 中，需要和 OMX 进行通信，跟踪代码，其中有一个变量 OMXClient mClient，位于 frameworks\av\media\libstagefright\OMXClient.cpp 中，代码如下：

```
OMXClient::OMXClient() {
}
struct MuxOMX : public IOMX {
    MuxOMX(const sp<IOMX> &remoteOMX);
    virtual ~MuxOMX();
    virtual IBinder *onAsBinder() { return IInterface::asBinder (mRemoteOMX).
get(); }
    //省略部分代码
private:
    mutable Mutex mLock;
    sp<IOMX> mRemoteOMX;
    sp<IOMX> mLocalOMX;
    const sp<IOMX> &getOMX(node_id node) const;
    const sp<IOMX> &getOMX_l(node_id node) const;
    //省略部分代码
};
```

OMXClient 有一个 IOMX 的变量 mOMX，它负责与 OMX 服务进行 Binder 通信。在 AwesomePlayer 的构造函数中会调用：

```
CHECK_EQ(mClient.connect(), (status_t)OK);
```

OMXClient 中的代码如下：

```
OMXClient::OMXClient() {
}
status_t OMXClient::connect() {
    sp<IServiceManager> sm = defaultServiceManager();
    sp<IBinder> binder = sm->getService(String16("media.player"));
    sp<IMediaPlayerService> service = interface_cast<IMediaPlayer Service>
(binder);
```



```
//省略部分代码
    mOMX = service->getOMX(); //获取 OMX
    if (!mOMX->livesLocally(0 /* node */, getpid())) {
        ALOGI("Using client-side OMX mux.");
        mOMX = new MuxOMX(mOMX);
    }
    return OK;
}

void OMXClient::disconnect() {
    if (mOMX.get() != NULL) {
        mOMX.clear();
        mOMX = NULL;
    }
}
```

OMXClient::connect 函数是通过 Binder 机制获得 MediaPlayerService 的，然后通过 MediaPlayerService 来创建 OMX 的实例。这样 OMXClient 就获得了 OMX 的入口，接下来可以通过 Binder 机制获得 OMX 提供的服务。也就是说，OMXClient 是 Android 中 OpenMAX 的入口。

在创建音视频解码 mVideoSource、mAudioSource 的时候，会把 OMXClient 中的 sp<IOMX> mOMX 的实例传给 mVideoSource、mAudioSource，来共享使用这个 OMX 的入口。也就是说，一个 AwesomePlayer 对应着一个 IOMX 变量，AwesomePlayer 中的音视频解码器共用这个 IOMX 变量来获得 OMX 服务。下面的代码是 AwesomePlayer 创建 OMX 的过程：

```
sp<IOMX> interface() {
    return mOMX;
}

status_t AwesomePlayer::initVideoDecoder(uint32_t flags) {
    ATRACE_CALL();
    bool setProtectionBit = false;
    char value[PROPERTY_VALUE_MAX];
    ALOGV("initVideoDecoder flags=0x%x", flags);
    //1
    mVideoSource = OMXCodec::Create(
        mClient.interface(), mVideoTrack->getFormat(),
        false, //createEncoder
        mVideoTrack,
        NULL, flags, USE_SURFACE_ALLOC ? mNativeWindow : NULL);
    if (mVideoSource != NULL) {
        int64_t durationUs;
        //省略部分代码
    }
}
```




```
//2
status_t err = mVideoSource->start();
//省略部分代码
}
//省略部分代码
return mVideoSource != NULL ? OK : UNKNOWN_ERROR;
}
```

以上分为 1、2 两个步骤，先看下 1 中的 OMXCodec::Create 函数：

```
sp<MediaSource> OMXCodec::Create(
    const sp<IOMX> &omx,
    const sp<MetaData> &meta, bool createEncoder,
    const sp<MediaSource> &source,
    const char *matchComponentName,
    uint32_t flags,
    const sp<ANativeWindow> &nativeWindow) {
    int32_t requiresSecureBuffers;
    const char *mime;
    bool success = meta->findCString(kKeyMIMETYPE, &mime);
    CHECK(success);
    Vector<String8> matchingCodecs;
    Vector<uint32_t> matchingCodecQuirks;
    findMatchingCodecs(
        mime, createEncoder, matchComponentName, flags,
        &matchingCodecs, &matchingCodecQuirks);
    //根据 mVideoTrack 传入的视频格式信息，查找相匹配的解码器
    sp<OMXCodecObserver> observer = new OMXCodecObserver;
    IOMX::node_id node = 0;
    //创建 OMXCodecObserver 实例，后续会详细介绍 OMXCodecObserver 的功能
    for (size_t i = 0; i < matchingCodecs.size(); ++i) {
        const char *componentNameBase = matchingCodecs[i].string();
        uint32_t quirks = matchingCodecQuirks[i];
        const char *componentName = componentNameBase;
        AString tmp;
        status_t err = omx->allocateNode(componentName, observer, &node);
        //通过 OMX 入口，依靠 Binder 机制调用 OMX 服务中的 allocateNode 函数，这一步一并传
        //入匹配得到的解码器组件名、OMXCodecObserver 实例和初始化为 0 的节点
        //这一步实际上分配了一个对解码器的控制节点，后续对解码器的操作都依靠这个节点
        if (err == OK) {
            ALOGV("Successfully allocated OMX node '%s'", componentName);
            sp<OMXCodec> codec = new OMXCodec(
                omx, node, quirks, flags,
                createEncoder, mime, componentName,
                source, nativeWindow);
        }
    }
}
```



```
        observer->setCodec(codec);
        err = codec->configureCodec(meta);
        //配置及初始化解码器
        if (err == OK) {
            if(!strcmp("OMX.Nvidia.mpeg2v.decode", componentName)) {
                codec->mFlags |= kOnlySubmitOneInputBufferAtOneTime;
            }
            return codec;
        }
        ALOGV("Failed to configure codec '%s'", componentName);
    }
}
return NULL;
}
```

每个 AwesomePlayer 实例只有一个 OMX 服务的入口,但是 AwesomePlayer 不一定只有一种解码器。音频、视频都有对应的解码器,部分场景下还有多路音频或者多路视频。这个时候 OMX 需要建立不同的解码器组件来应对 AwesomePlayer 中不同的解码需求。

OMX 中有非常重要的两个成员 OMXMaster 和 OMXNodeInstance。OMX 通过这两个成员来创建和维护不同的 OpenMAX 解码器组件,为 AwesomePlayer 中不同的解码需求提供服务。OMXNodeInstance 负责创建并维护不同的实例,这些实例是根据实际的解码需求创建的,以 node_id 作为唯一标识。

这样解码器组件中每个 OMXCodec 在 OMX 服务器端都有了自己对应的 OMXNodeInstance 实例。AwesomePlayer 就可以根据这个 OMXNodeInstance 来操作相应的解码器。

OMXMaster 维护底层软硬件解码库,用于管理解码器组件,根据 OMXNodeInstance 中想要的解码器来创建解码实体组件。所以我们要追踪一下 OMXMaster 和 OMXNodeInstance。

在 OMX 构造函数中会进行初始化操作。OMX.cpp 中的代码如下:

```
OMX::OMX()
    : mMaster(new OMXMaster.),
      mNodeCounter(0) {
}
```

OMXMaster.cpp 中的代码如下:

```
OMXMaster::OMXMaster()
    : mVendorLibHandle(NULL) {
    addVendorPlugin();
    addPlugin(new SoftOMXPlugin);
}
```




```
OMXMaster::~~OMXMaster() {
    clearPlugins();

    if (mVendorLibHandle != NULL) {
        dlclose(mVendorLibHandle);
        mVendorLibHandle = NULL;
    }
}

void OMXMaster::addVendorPlugin() {
    addPlugin("libstagefrighthw.so");
}

void OMXMaster::addPlugin(const char *libname) {
    mVendorLibHandle = dlopen(libname, RTLD_NOW);

    if (mVendorLibHandle == NULL) {
        return;
    }

    typedef OMXPluginBase *(*CreateOMXPluginFunc)();
    CreateOMXPluginFunc createOMXPlugin =
        (CreateOMXPluginFunc)dlsym(
            mVendorLibHandle, "createOMXPlugin");
    if (!createOMXPlugin)
        createOMXPlugin = (CreateOMXPluginFunc)dlsym(
            mVendorLibHandle, "_ZN7android15createOMXPluginEv");
    if (createOMXPlugin) {
        addPlugin((*createOMXPlugin)());
    }
}

void OMXMaster::addPlugin(OMXPluginBase *plugin) {
    Mutex::Autolock autoLock(mLock);

    mPlugins.push_back(plugin);
    OMX_U32 index = 0;
    char name[128];
    OMX_ERRORTYPE err;
    while ((err = plugin->enumerateComponents(
        name, sizeof(name), index++)) == OMX_ErrorNone) {
        String8 name8(name);

        if (mPluginByComponentName.indexOfKey(name8) >= 0) {
```




```
        ALOGE("A component of name '%s' already exists, ignoring
this one.", name8.string());
        continue;
    }
    mPluginByComponentName.add(name8, plugin);
}

if (err != OMX_ErrorNoMore) {
    ALOGE("OMX plugin failed w/ error 0x%08x after registering %zu "
        "components", err, mPluginByComponentName.size());
}
}

void OMXMaster::clearPlugins() {
    Mutex::Autolock autoLock(mLock);
    typedef void (*DestroyOMXPluginFunc)(OMXPluginBase*);
    DestroyOMXPluginFunc destroyOMXPlugin =
        (DestroyOMXPluginFunc)dlsym(
            mVendorLibHandle, "destroyOMXPlugin");
    mPluginByComponentName.clear();
    for (List<OMXPluginBase*>::iterator it = mPlugins.begin();
        it != mPlugins.end(); ++it) {
        if (destroyOMXPlugin)
            destroyOMXPlugin(*it);
        else
            delete *it;
        *it = NULL;
    }

    mPlugins.clear();
}
```

到这里，就可以明白 AwesomePlayer 是如何利用具体硬件平台上的硬件解码器的。

那么针对不同的文件格式，如何选择具体的解码器组件呢？继续顺着前面的 OMXCodec::Create 介绍，看一下 allocateNode 函数，其在 OMX.cpp 中的代码如下：

```
status_t OMX::allocateNode(const char *name, const sp<IOMXObserver>
&observer, node_id *node) {
    Mutex::Autolock autoLock(mLock);
    *node = 0;
    OMXNodeInstance *instance = new OMXNodeInstance(this, observer, name);
    OMX_COMPONENTTYPE *handle;
    OMX_ERRORTYPE err = mMaster->makeComponentInstance(
        name, &OMXNodeInstance::kCallbacks,
```



```
        instance, &handle);
    if (err != OMX_ErrorNone) {
        ALOGE("FAILED to allocate omx component '%s' err=%s(%#x)", name,
asString(err), err);
        instance->onGetHandleFailed();
        return StatusFromOMXError(err);
    }

    *node = makeNodeID(instance);
    mDispatchers.add(*node, new CallbackDispatcher(instance));
    instance->setHandle(*node, handle);
    mLiveNodes.add(IInterface::asBinder(observer), instance);
    IInterface::asBinder(observer)->linkToDeath(this);
    return OK;
}
```

以上代码实例化了 `OMXNodeInstance`，接下来会调用 `OMXNodeInstance` 的 `makeComponentInstance` 函数，代码如下：

```
OMX_ERRORTYPE OMXMaster::makeComponentInstance(
    const char *name,
    const OMX_CALLBACKTYPE *callbacks,
    OMX_PTR appData,
    OMX_COMPONENTTYPE **component) {
    Mutex::Autolock autoLock(mLock);
    *component = NULL;
    ssize_t index = mPluginByComponentName.indexOfKey(String8(name));
    if (index < 0) {
        return OMX_ErrorInvalidComponentName;
    }

    OMXPluginBase *plugin = mPluginByComponentName.valueAt(index);
    OMX_ERRORTYPE err = plugin->makeComponentInstance(name, callbacks,
appData, component);
    if (err != OMX_ErrorNone) {
        return err;
    }

    mPluginByInstance.add(*component, plugin);
    return err;
}
```

这样就实现了根据文件编码格式对具体解码器的连接。接下来了解一下 `OMXCodec` 如何注册和初始化 `OMX` 所需要的回调函数。



OMX 服务主要完成 3 个任务：NodeInstance 列表的管理、NodeInstance 的操作、事件的处理。

6.1.3 OMX 中 NodeInstance 列表的管理

在我们创建 ComponentInstance (OMX 组件实例) 后，需要对它里面的 NodeInstance 列表进行管理。

- OMX 对解码器组件 Component 的使用，是通过 OMXNodeInstance 来实现的。OMXNodeInstance 自身的动作包括 NodeInstance 的生成 (allocateNode) 和删除 (freeNode)。其实就是对 mDispatchers 和 mNodeIDToInstance 进行添加和删除。
- mNodeIDToInstance 就是一个 key 为 node_id, value 为 NodeInstance 的键/值对列表。而 mDispatchers 就是一个 key 为 node_id, value 为 OMX::CallbackDispatcher 的键/值对列表。并且，每个 NodeInstance 都拥有一个 OMX::CallbackDispatcher。
- CallbackDispatcher 的主要作用是在解码器组件 Component 发出回调动作后，将 message 分发给对应的 OMXCodec 客户端。

6.1.4 OMX 中 NodeInstance 节点的操作

在我们了解到 OMXNodeInstance 列表管理后，每个 OMXNodeInstance 中都有 Node 节点，若你需要给这些 Node 节点分配一些 Buffer，下面看看对 Node 节点的操作过程。

OMXNodeInstance 主要的成员函数如下：

```
struct OMXNodeInstance {
    OMXNodeInstance(
        OMX *owner, const sp<IOMXObserver> &observer);
    void setHandle(OMX::node_id node_id, OMX_HANDLETYPE handle);
    status_t prepareForAdaptivePlayback(
        OMX_U32 portIndex, OMX_BOOL enable,
        OMX_U32 maxFrameWidth, OMX_U32 maxFrameHeight);
    status_t useBuffer(
        OMX_U32 portIndex, const sp<IMemory> &ms,
        OMX::buffer_id *buffer);
    //OMX::Client 通过此函数将已分配好的 Buffer 传给 OMX 服务器端组件，让其使用
    status_t allocateBuffer(
        OMX_U32 portIndex, size_t size, OMX::buffer_id *buffer,
        void **buffer_data);
    //Client 通过调用此函数让 Component 分配 Buffer
    status_t allocateBufferWithBackup(
        OMX_U32 portIndex, const sp<IMemory> &ms,
```



```

        OMX::buffer_id *buffer);
status_t freeBuffer(OMX_U32 portIndex, OMX::buffer_id buffer);
//Client 通过调用此函数让 Component 释放 allocateBuffer 分配的 Buffer
status_t fillBuffer(OMX::buffer_id buffer);
//Client 通过调用此函数传递空的 Buffer 给 Component, 让其将处理好的
//数据填入其中。此函数会调用 OMX 标准接口 OMX_FillThisBuffer
status_t emptyBuffer(
    OMX::buffer_id buffer,
    OMX_U32 rangeOffset, OMX_U32 rangeLength,
    OMX_U32 flags, OMX_TICKS timestamp); //Client 通过调用此函数传递
//输入 Buffer 给 Component, 让其读取其中的数据并进行编解码等处理操作。此函数会调用
//OMX 标准接口 OMX_EmptyThisBuffer
//省略部分代码
static OMX_ERRORTYPE OnEmptyBufferDone(
    OMX_IN OMX_HANDLETYPE hComponent,
    OMX_IN OMX_PTR pAppData,
    OMX_IN OMX_BUFFERHEADERTYPE *pBuffer); //在 Component 完成对输入
//Buffer 的读取后, 调用此回调函数, 向 Client 发送 EmptyBufferDone 消息
static OMX_ERRORTYPE OnFillBufferDone(
    OMX_IN OMX_HANDLETYPE hComponent,
    OMX_IN OMX_PTR pAppData,
    OMX_IN OMX_BUFFERHEADERTYPE *pBuffer);
//在 Component 完成相应的处理操作并将输出数据填入输出 Buffer 后, 调用此回调
//函数, 向 Client 发送 FillBufferDone 消息
status_t storeMetaDataInBuffers_l(OMX_U32 portIndex, OMX_BOOL enable);
sp<GraphicBufferSource> getGraphicBufferSource();
void setGraphicBufferSource(const sp<GraphicBufferSource>& bufferSource);
OMXNodeInstance(const OMXNodeInstance &);
OMXNodeInstance &operator=(const OMXNodeInstance &);
};

```

当执行这些函数时, 都是先通过 `findInstance` 在 `mNodeIDToInstance` 列表中找到对应的 `NodeInstance`, 然后调用 `NodeInstance` 对应的方法。

OMXCodec 对具体的 component 函数的操作, 是通过 `OMXNodeInstance` 来实现的, 如 `fillBuffer`、`emptyBuffer`、`sendCommand` 等, 它们都是通过 `OMX_Core.h` 中的宏定义间接调用 `OMX_Component.h` 的 `OMX_COMPONENTTYPE` 中的相应函数指针来完成的。`OMX_Core.h` 和 `OMX_Component.h` 都是 OpenMAX 标准头文件。

在 `OMXNodeInstance.cpp` 中有这样一段代码:

```

OMX_CALLBACKTYPE OMXNodeInstance::kCallbacks = {
    &OnEvent, &OnEmptyBufferDone, &OnFillBufferDone
};

```

它把 3 个 OMXNodeInstance 类的静态方法注册给了 kCallbacks。kCallbacks 实际就是 struct OMX_COMPONENTTYPE 和 struct OMX_CALLBACKTYPE 的具体实现，而这两者就是在 OMX_Core.h 和 OMX_Component.h 中定义的。

在哪里使用 kCallbacks 呢？下面看看 OMX.cpp 中的 allocateNode 函数：

```
status_t OMX::allocateNode(const char *name, const sp<IOMXObserver>
&observer, node_id *node) {
    Mutex::Autolock autoLock(mLock);
    *node = 0;
    OMXNodeInstance *instance = new OMXNodeInstance(this, observer, name);
    OMX_COMPONENTTYPE *handle;
    OMX_ERRORTYPE err = mMaster->makeComponentInstance(
        name, &OMXNodeInstance::kCallbacks,
        instance, &handle);
    if (err != OMX_ErrorNone) {
        ALOGE("FAILED to allocate omx component '%s' err=%s(%#x)", name,
asString(err), err);
        instance->onGetHandleFailed();
        return StatusFromOMXError(err);
    }
    *node = makeNodeID(instance);
    mDispatchers.add(*node, new CallbackDispatcher(instance));
    instance->setHandle(*node, handle);
    mLiveNodes.add(IInterface::asBinder(observer), instance);
    IInterface::asBinder(observer)->linkToDeath(this);
    return OK;
}
```

事件处理函数传给了组件 ComponentInstance，也就是传给了具体芯片平台相关的 OMX IL 层。

当组件有事件发生时，就会调用 OMXNodeInstance 中这几个注册过的事件处理函数：

```
OMX_ERRORTYPE OMXNodeInstance::OnEmptyBufferDone
OMX_ERRORTYPE OMXNodeInstance::OnFillBufferDone
OMX_ERRORTYPE OMXNodeInstance::OnEvent
```

而这几个函数又会调用 OMX 中对应的函数，也就是下面这 3 个函数：

```
OMX_ERRORTYPE OMX::OnEmptyBufferDone
OMX_ERRORTYPE OMX::OnFillBufferDone
OMX_ERRORTYPE OMX::OnEvent
```

总结一下，这几个函数都采用相同的方式，即根据 node_id 找到 CallbackDispatcher，并把事件信息传递出去，也就是 findDispatcher(node)->post(msg)。

进一步地，需要了解 CallbackDispatcher 的实现机制。它内部开启了一个线程，使用了信号量（signal）机制。

findDispatcher(node)->post(msg)是一个异步操作，只把消息传递过去，不会等待事件处理完毕就返回。

问题来了，那么 CallbackDispatcher 是怎么处理接收到的消息的呢？查看以下代码：

```
bool OMX::CallbackDispatcher::loop() {
    for (;;) {
        omx_message msg;
        {
            Mutex::Autolock autoLock(mLock);
            while (!mDone && mQueue.empty()) {
                mQueueChanged.wait(mLock);
            }
            if (mDone) {
                break;
            }
            msg = *mQueue.begin();
            mQueue.erase(mQueue.begin());
        }
        dispatch(msg);
    }
    return false;
}
```

这样事件最终还是跨 Binder 传到 OMXCodec 里面，交给 OMXCodecObserver 了。一旦有关于回调的过程，再从 OMX 服务器端发送到 OMX 客户端。我们又知道 AwesomePlayer 类中持有 OMX 客户端，所以这些从 OMX 组件通知上来的消息就可以到达 AwesomePlayer 中。这样就完成了 AwesomePlayer 和 OMX 组件之间的通信。

6.1.5 总结 AwesomePlayer 到 OMX 服务过程

最后，总结一下前面的内容。

- 在 AwesomePlayer 初始化过程中，通过 initVideoDecoder/initAudioDecoder 函数来创建音视频解码器 mVideoSource/mAudioSource。
- 在 mVideoSource 中通过 mVideoTrack 来解复用媒体文件，从中获取文件编码格式，继而得到需要的解码器类型，通过类型调用 omx->allocateNode 创建 OMX node 实例，与编码格式对应。以后都是通过 Node 实例来操作实际的硬件解码器的。
- 初始化 MediaPlayerService 对象的时候会创建 OMX 对象，OMX 对象的构造函数会创建

mMaster, mMaster 负责获得与管理硬件平台的硬件解码器组件库。

- 在 omx->allocateNode 中通过 mMaster->makeComponentInstance 来创建真正对应的解码器组件。这个解码器组件将完成之后实质的解码工作。
- 在创建 mMaster->makeComponentInstance 的过程中, 通过上面 mVideoTrack 传递过来的解码器类型名, 找到相对应的解码器的库, 然后实例化。
- 解码 Component 通过输入 Port 和输出 Port 进行交互, 通过和 OMXCodec 共享 Buffer 进行编解码。
- AwesomePlayer 包含了 mVideoSource, 当初始化时指向 OMXCodec 的实际对象。OMXCodec 使用了 Binder 机制, 实现了对 OMX 服务的远程调用, 其中 IOMX 作为接口类定义了 OMX 的大部分接口函数。
- 当具体实现 OMX 时, OMXMaster 类用于管理 OMX 的插件, OMXNodeInstance 类代表 OMX 的具体实例, 完成和 Component 的调用和交互。
- CallbackDispatcher 用于调度处理回调函数传回的消息。OMXNodeInstance 和 CallbackDispatcher 一一对应, 协同工作, 完成不同实例的消息处理。
- OMXNodeInstance 是 OMX 端的概念, 是服务器端的概念。其服务器端与 OMX 在一个进程空间中。
- OMXObserver 是 OMXCodec 端的概念, 是客户端的概念。其客户端与 OMXCodec 在一个进程空间中。其 Bn、Bp 方向和 OMX、OMXNodeInstance 相反, 主要用于反向通知 onMessage 消息。

到此, 就介绍完 AwesomePlayer 是如何对 OMX 进行初始化的, 以及如何关联到对应硬件平台上的 HardWare 解码器的。

6.2 Codec 部分中的 OMXCodec 与 OMX 事件回调流程

6.1 节介绍了 AwesomePlayer 到 OMX 服务, 其中提及 OMX 服务主要完成 3 个任务, 分别是 NodeInstance 列表的管理、NodeInstance 的操作、事件的处理。最后这个事件的处理就是放大要看的內容。要一步一步进行编解码操作, 事件传递进行通信是必不可少的环节, 下面看看将要介绍的知识点。

- OMXCodec 与 OMX callback 事件的处理时序图。
- 如何从 OMX 中将事件分发到 OMXCodec (附时序图)。
- 缓冲区更新过程。
- 消息回调。

6.2.1 OMXCodec 与 OMX callback 事件的处理时序图

OMXCodec 与 OMX callback 事件的处理时序图如图 6-3 所示。

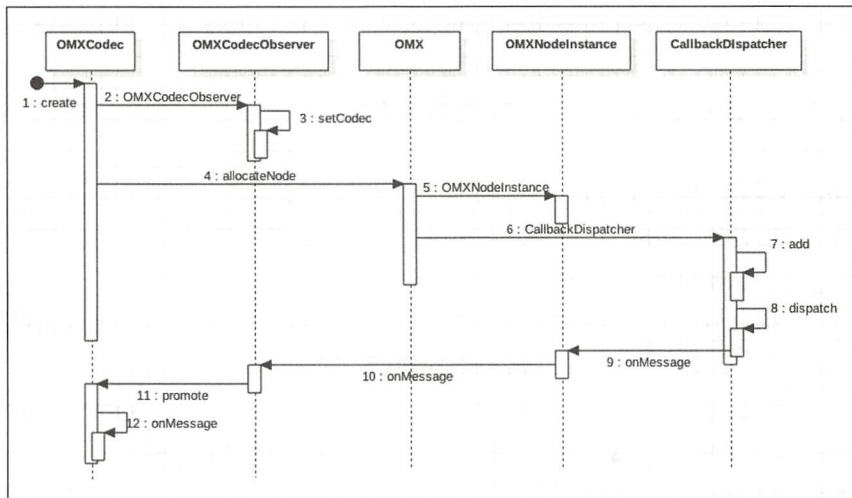


图 6-3 OMXCodec 与 OMX callback 事件的处理时序图

从如图 6-3 所示的时序图来看，首先要建立一个 OMXCodecObserver 类，该类是 OMXCodec 的内部类，在 create 函数中被创建，并把对应的 OMXCodec 加入自己的观察范围内，具体代码（framework/base/media/libstagefright/OMXCodec.cpp）如下：

```

sp<MediaSource> OMXCodec::Create(
    const sp<IOMX> &omx,
    const sp<MetaData> &meta, bool createEncoder,
    const sp<MediaSource> &source,
    const char *matchComponentName,
    uint32_t flags) {
    //省略部分代码
    sp<OMXCodecObserver> observer = new OMXCodecObserver;
    //省略部分代码
    observer->setCodec(codec);
    //省略部分代码
}
  
```

其次初始化它的 callback 事件和事件的派发处理函数。那么 OMX 主要的 callback 事件有哪些呢？看看 framework/base/media/libstagefright/omx/OMXNodeInstance.cpp 的 kCallbacks 函数中的定义：

```

OMX_CALLBACKTYPE OMXNodeInstance::kCallbacks = {
  
```

```

    &OnEvent, &OnEmptyBufferDone, &OnFillBufferDone
};

```

callback 在哪定义呢？看看 framework/base/media/libstagefright/omx/OMX.cpp 中的代码：

```

status_t OMX::allocateNode(
    //省略部分代码
    OMXNodeInstance *instance = new OMXNodeInstance(this, observer);
    OMX_COMPONENTTYPE *handle;
    OMX_ERRORTYPE err = mMaster->makeComponentInstance(
        name, &OMXNodeInstance::kCallbacks,
        instance, &handle);
    //省略部分代码
    mDispatchers.add(*node, new CallbackDispatcher(instance));
    //省略部分代码
}

```

即每个 Component（组件）对应一组回调事件。这些（回调事件）由哪些函数返回呢？具体的定义在 framework/base/media/libstagefright/openmax/OMX_Core.h 中：

```

callback EventHandler()
#define OMX_SendCommand(hComponent, Cmd, nParam, pCmdData)
    ((OMX_COMPONENTTYPE*)hComponent)->SendCommand(
        hComponent, Cmd, nParam, pCmdData)
EmptyBufferDone call back.
#define OMX_EmptyThisBuffer(hComponent, pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->EmptyThisBuffer(
        hComponent, pBuffer) /* Macro End */
FillBufferDone call back
#define OMX_FillThisBuffer(
    hComponent, pBuffer)
    ((OMX_COMPONENTTYPE*)hComponent)->FillThisBuffer(
        hComponent, pBuffer) /* Macro End */

```

有了 callback 事件，如何 dispatch 呢？其实我们在 allocateNote 函数中已经定义好 dispatch 的函数 mDispatchers.add(*node, new CallbackDispatcher(instance)); 了。

6.2.2 如何从 OMX 中分发事件到 OMXCodec

有了 Observer（观察者）、Callback Event（事件回调）、CallbackDispatcher（事件回调分发者），那么一个事件如何从 OMX 分发到 OMXCodec 呢？

下面以 EmptyBuffer 流程来具体看看，时序图如图 6-4 所示。

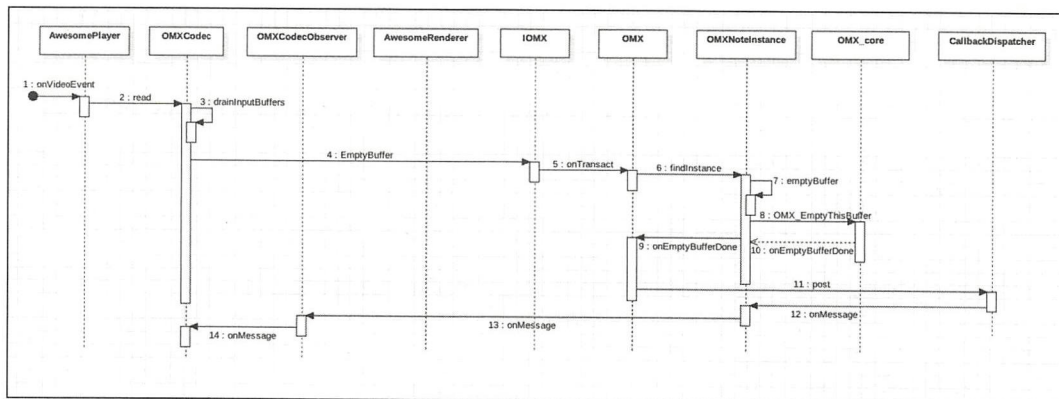


图 6-4 EmptyBuffer 流程时序图

首先查看 `mVideoSource->read`，实际上就是调用 `OMXCodec::read` 函数，代码如下：

```

status_t OMXCodec::read( MediaBuffer **buffer, const ReadOptions *options)
{
    if (mInitialBufferSubmit) {
        mInitialBufferSubmit = false;    //初始化值是 true，这里置为 false，是
        //为了保证初始化过的变量不再进入下面的代码逻辑，即只初始化一次
        if (seeking) {
            CHECK(seekTimeUs >= 0);
            mSeekTimeUs = seekTimeUs;
            mSeekMode = seekMode;
            //没有理由触发下面的代码，没有任何东西可以刷新
            seeking = false;
            mPaused = false;
        }
        drainInputBuffers(); //调用 drainInputBuffers 函数，把输入通道中的所
        //有输入缓冲区逐个传递给 drainInputBuffer，即先把 inputbuffer 都读满，然后一次性发
        //送给具体的 Component，让其慢慢解码，drainInputBuffer 的实现见后面的内容
        if (mState == EXECUTING) {
            //如果 mState == RECONFIGURING，此代码将在重新启用输出端口后触发
            fillOutputBuffers();
        }
    }
    while (mState != ERROR && !mNoMoreOutputData && mFilledBuffers.
empty()) {
        if ((err = waitForBufferFilled_1()) != OK) {
            return err;
        }
    }
    //等待输出缓冲区的数据，如果有数据就往下执行，读取数据输出
    //等到 OMXCodec::read 进行读取后，就直接到这一步，等待输出缓冲区数据

```

```

//这个读取消息主要在 omx_message::FILL_BUFFER_DONE 处理流程中发送, 这时 OpenMAX
//在解码数据后把一个输出缓冲区填满, 然后触发这个 FILL_BUFFER_DONE 事件
//在 omx_message::FILL_BUFFER_DONE 处理流程中有下面两行代码
//
//          mFilledBuffers.push_back(i);
//          mBufferFilled.signal();
//即把已经填充好的输出缓冲区索引保存到 mFilledBuffers 中, 然后发送消息
//          size_t index = *mFilledBuffers.begin();
//          mFilledBuffers.erase(mFilledBuffers.begin()); //检查到有数据可读, 就从
//mFilledBuffers 读取头部的缓冲区索引, 同时从 List 中删除这个索引
//          BufferInfo *info = &mPortBuffers[kPortIndexOutput].editItemAt(index);
//          CHECK_EQ((int)info->mStatus, (int)OWNED_BY_US);
//          info->mStatus = OWNED_BY_CLIENT;
//          info->mMediaBuffer->add_ref();
//然后根据索引找到缓冲区, 再把缓冲区地址赋值给输出指针进行输出, 这个缓冲区的引用计数会
//加1, 上层使用完会释放
//          *buffer = info->mMediaBuffer;
//          return OK;
}

```

在上面的代码注释中, 简单介绍了 `drainInputBuffers`, 那么下面看看 `drainInputBuffer` 的实现代码, 如下:

```

bool OMXCodec::drainInputBuffer(BufferInfo *info)
{
    MediaBuffer *srcBuffer;
    err = mSource->read(&srcBuffer, &options);
    //在 drainInputBuffer 中会通过调用 mSource->read 读取原始数据, 填充到缓冲区中, 然
    //后调用 mOMX->emptyBuffer 发送消息给 OpenMAX Component
    err = mOMX->emptyBuffer(
        mNode, info->mBuffer, 0, offset,
        flags, timestampUs);
}

void OMXCodec::fillOutputBuffer(BufferInfo *info) {
    status_t err = mOMX->fillBuffer(mNode, info->mBuffer);
    //调用 OMXCodec 中的 fillOutputBuffer 函数的作用是把输出通道中的输出缓冲区逐个传递
    //给输出 Buffer 在 OMXCodec 中的 fillOutputBuffer 函数调用 mOMX->fillBuffer 发送消息
    //给 OpenMAX, 相当于把缓冲区传递给 OpenMAX
}

```

总结一下上面的代码, 这是第一次执行 `OMXCodec::read` 时的操作, 当整个编解码流程运行起来之后, 会面临一个输入/输出缓冲区更新的问题。

6.2.3 缓冲区更新过程

输入缓冲区更新过程为, 如果一个输入缓冲区数据被读取完, OpenMAX 就触发事件

omx_message::EMPTY_BUFFER_DONE 通知上层，在这个事件处理流程中，会根据发送来的 bufferid 找到对应的输入缓冲区，然后把这个缓冲区传递给 drainInputBuffer，继续往下执行，代码如下：

```
void OMXCodec::on_message(const omx_message &msg) {
    case omx_message::EMPTY_BUFFER_DONE:
    {
        drainInputBuffer(&buffers->editItemAt(i));
        break;
    }
}
```

输出缓冲区更新过程如下：解码完毕后，OpenMAX 组件触发 omx_message::FILL_BUFFER_DONE，输出缓冲区会被传出交给上层使用（传递给 SurfaceFlinger 来显示），使用完后需要把这个缓冲区重新交给 OpenMAX。上层使用完输出缓冲区后会调用 MediaBuffer::release 销毁缓冲区，在这个接口中会给输出缓冲区的引用计数减 1。然后调用 signalBufferReturned，实际对应 OMXCodec::signalBufferReturned 接口。最后，下一层调用 fillOutputBuffer 函数，把这个缓冲区重新交给 OpenMAX。

综上所述，这样就通过第一次调用 drainInputBuffer 触发了 OpenMAX，然后依靠 OpenMAX 的事件驱动来完成数据的读取、解码操作。

下面以输出缓冲区为例，再具体介绍一下上面的分析流程。OpenMAX Component 解码完 1 帧之后，会调用 ppCallbacks->FillBufferDone，也就是调用了之前初始化好的 OMX::OnFillBufferDone：

```
OMX_ERRORTYPE OMX::OnFillBufferDone(
    node_id node, OMX_IN OMX_BUFFERHEADERTYPE *pBuffer) {
    ALOGV("OnFillBufferDone buffer=%p", pBuffer);
    omx_message msg;
    msg.type = omx_message::FILL_BUFFER_DONE;
    //设置消息的类型，便于接收端进行辨识
    msg.node = node;
    msg.u.extended_buffer_data.buffer = pBuffer;
    msg.u.extended_buffer_data.range_offset = pBuffer->nOffset;
    msg.u.extended_buffer_data.range_length = pBuffer->nFilledLen;
    msg.u.extended_buffer_data.flags = pBuffer->nFlags;
    msg.u.extended_buffer_data.timestamp = pBuffer->nTimeStamp;
    msg.u.extended_buffer_data.platform_private = pBuffer->
pPlatformPrivate;
    msg.u.extended_buffer_data.data_ptr = pBuffer->pBuffer;
    findDispatcher(node)->post(msg);
    //将 msg 信息打包，然后通过 node 找到之前注册好的 Dispatcher，把 msg post 出去
```



```

    return OMX_ErrorNone;
}

```

接着看看 post 函数:

```

void OMX::CallbackDispatcher::post(const omx_message &msg) {
    Mutex::Autolock autoLock(mLock);
    mQueue.push_back(msg);
    mQueueChanged.signal(); //将 msg 放入队列, 触发信号
}

bool OMX::CallbackDispatcher::loop() { //前面介绍过, OMX::allocateNode 的作用
//主要是分配节点和 Buffer 空间, 这里创建了一个回调分发消息的轮循, 不断轮循看是否有新的消息
    for (;;) {
        omx_message msg;
        {
            Mutex::Autolock autoLock(mLock);
            while (!mDone && mQueue.empty()) {
                mQueueChanged.wait(mLock); //队列被唤醒
            }
            if (mDone) {
                break;
            }
            msg = *mQueue.begin();
            mQueue.erase(mQueue.begin()); //获取最新发送过来的消息
        }
        dispatch(msg); //调用 CallbackDispatcher::dispatch 函数, 分发消息
    }
    return false;
}

```

接下来把这个消息分发出去:

```

void OMX::CallbackDispatcher::dispatch(const omx_message &msg) {
    if (mOwner == NULL) {
        ALOGV("Would have dispatched a message to a node that's already
gone.");
        return;
    }
    mOwner->onMessage(msg);
}

```

6.2.4 消息回调

在消息回调中, 主要通过 onMessage 函数处理消息, 然后回调给上面调用 OMX 接口的类, 代码如下:

```

void OMXNodeInstance::onMessage(const omx_message &msg) {
    if (msg.type == omx_message::FILL_BUFFER_DONE) {
        OMX_BUFFERHEADERTYPE *buffer =
            static_cast<OMX_BUFFERHEADERTYPE *>(
                msg.u.extended_buffer_data.buffer);
        BufferMeta *buffer_meta =
            static_cast<BufferMeta *>(buffer->pAppPrivate);
        buffer_meta->CopyFromOMX(buffer);
    } else if (msg.type == omx_message::EMPTY_BUFFER_DONE) {
        const sp<GraphicBufferSource>& bufferSource(getGraphicBufferSource());
        if (bufferSource != NULL) {
            OMX_BUFFERHEADERTYPE *buffer =
                static_cast<OMX_BUFFERHEADERTYPE *>(
                    msg.u.buffer_data.buffer);
            bufferSource->codecBufferEmptied(buffer);
            return;
        }
    }
    mObserver->onMessage(msg);
}

```

通过调用 OMXCodecObserver 把消息通过 onMessage 函数传给 OMXCodec，代码如下：

```

virtual void onMessage(const omx_message &msg) {
    //经过跨 Binder 调用，最终调用到 OMXCodec 端，即 AwesomePlayer 的进程空间
    sp<OMXCodec> codec = mTarget.promote();
    if (codec.get() != NULL) {
        Mutex::Autolock autoLock(codec->mLock);
        codec->on_message(msg);
        codec.clear();
    }
}

void OMXCodec::on_message(const omx_message &msg) {
    switch (msg.type) {
        case omx_message::FILL_BUFFER_DONE:
        {
            //省略部分代码
            mFilledBuffers.push_back(i); //mFilledBuffers 的数据类型是
            //List<size_t>，它存储的不是缓冲区地址而是输出缓冲区索引。
            //把已经填充好的输出缓冲区索引保存到 mFilledBuffers 中，然后发送信号
            //那么谁等在这个 Condition（状态）上呢？原来 OMXCodec::read 函数的后半段中的
            //waitForBufferFilled_1 会一直等待
            //OMXCodec::read 函数中返回给 AwesomePlayer 的 mVideoBuffer，就是 OMX 框架中
            //outPutPort 上的 Buffer
            mBufferFilled.signal();
        }
    }
}

```

```

        if (mIsEncoder) {
            sched_yield();
        }
    }
}

```

通过上面的分析可以看到，虽然在 OMX 框架中输入/输出端口上的 Buffer（缓冲数据）的生产和消费是异步的，但还是通过一个 Condition mBufferFilled 来做了同步，通过信号量机制达到同步的目的。这本质上是一个生产者/消费者模型。对于 Codec（编解码）部分中的 OpenMAX 框架，可以参考《Android 系统级深入开发：移植与调试》一书中的 18 章，其中详细介绍了 OpenMAX 的相关知识，本书不再赘述。

提示：Android 中的 OpenMAX 适配层是 OpenMAX IL 层之上的封装层，在 Android 系统中被 StageFright 调用，也可以被其他部分调用。

6.3 MediaCodec 相关知识

在 Android 中，除了通过 MediaPlayer 播放视频，还可以通过 MediaCodec 播放视频，那么到底 MediaCodec 和 MediaPlayer 播放视频有什么不同呢？这就需要我们对 MediaCodec 进行深入了解。

6.3.1 MediaCodec 的基本认识

1. MediaCodec 是什么

MediaCodec 类可以访问底层媒体编解码器框架（StageFright 或 OpenMAX），即编解码组件。这是 Android low-level 多媒体支持基础设施的一部分（通常与 MediaExtractor、MediaSync、MediaMuxer、MediaCrypto、MediaDrm、Image、Surface 和 AudioTrack 一起使用）。它本身并不是 Codec，它通过调用底层编解码组件获得了 Codec 的能力。

2. 创建 MediaCodec 的方式

创建 MediaCodec（按格式创建）：

- MediaCodec createDecoderByType(String type)：创建解码器。
- MediaCodec createEncoderByType(String type)：创建编码器。
- type：数据解析阶段的 mimeType，如“video/avc”。

创建 MediaCodec（按 Codec 名字创建）：

Android 音视频开发

- MediaCodec createByCodecName(String name)。
- OMX.google.h264.decoder: 软解码。
- OMX.MTK.VIDEO.DECODER.AVC: 硬解码。

3. MediaCodec 的工作方式

MediaCodec 的工作方式如图 6-5 所示。

MediaCodec 处理输入数据产生输出数据。当异步处理数据时，使用一组输入和输出 Buffer 队列。通常，在逻辑上，客户端请求（或接收）数据后填入预先设定的空输入缓冲区，输入 Buffer 填满数据后将其传递到 MediaCodec 并进行编解码处理。之后 MediaCodec 编解码后的数据被填充到一个输出 Buffer 中。最后，客户端请求（或接收）输出 Buffer，消耗输出 Buffer 中的内容，用完释放，给回 MediaCodec 重新填充输出数据。

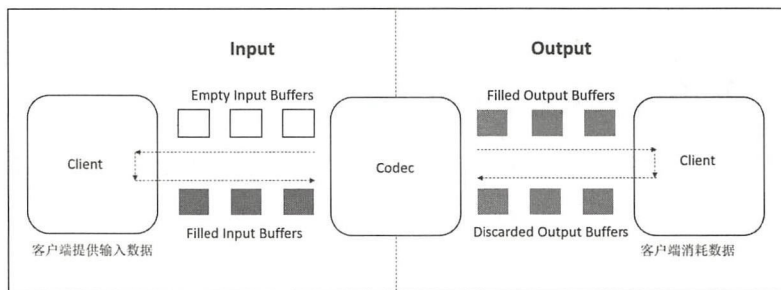


图 6-5 MediaCodec 的工作方式

需要注意的是，必须保证输入和输出队列同时非空，即至少持有一个输入 Buffer 和输出 Buffer 才能工作。

- MediaCodec 读取输入 Buffer 的数据，进行解码（或编码），将处理完的数据填入输出 Buffer。
- MediaCodec 处理完一个输入 Buffer 的数据，将此 Buffer 移入客户端队列，客户端可以继续使用。
- MediaCodec 将填充的输出 Buffer 放入客户端队列，客户端可以取出数据进行播放。

4. MediaCodec 状态周期图

在 MediaCodec 的生命周期内存在 3 种状态，即 Stopped、Executing 和 Released。Stopped 状态实际上还可处于 3 种状态，即 Uninitialized、Configured 和 Error，而 Executing 状态概念上的进展通过 3 个子状态进行，即 Flushed、Running 和 End-of-Stream。MediaCodec 状态周期图如图 6-6 所示。

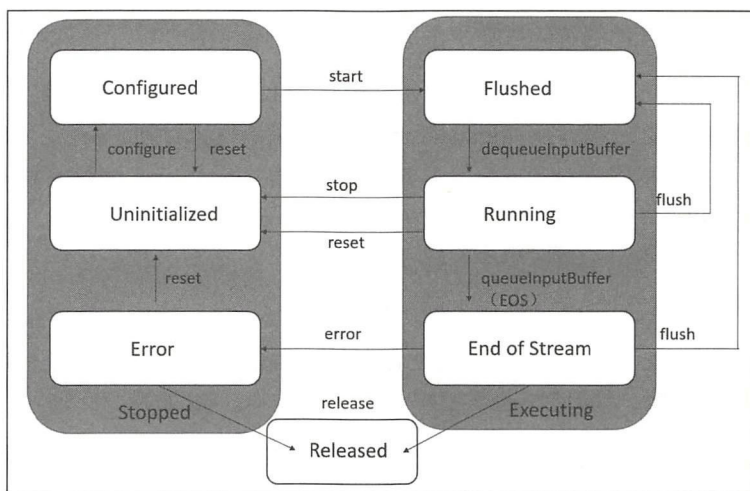


图 6-6 MediaCodec 状态周期图

当使用工厂方法创建一个 MediaCodec 时, MediaCodec 处于未初始化的状态。首先, 需要通过 `configure(...)` 配置它, 让它处于配置状态, 然后调用 `start` 函数将它转变为 Executing 状态。

Executing 状态有 3 个子状态, 分别是 Flushed、Running 和 End-of-Stream。在调用 `start` 函数后, MediaCodec 会立刻刷新子状态, 它拥有所有的 Buffer。一旦第一个输入 Buffer 被从列中移除, MediaCodec 将花费比较长的时间移动到正在运行的子状态上。当队列的输入 Buffer 带有 End-of-Stream 标记, MediaCodec 将转换到 End-of-Stream 子状态。在这种状态下的 MediaCodec 不再接受进一步输入 Buffer, 但仍然生成输出 Buffer, 直到达到 End-of-Stream 状态输出。在 Executing 状态的任何时候调用 `flush` 函数, 可以移回到 Flushed 子状态。

调用 `stop` 函数返回的 MediaCodec 处于 Uninitialized (未初始化) 状态, 因为它可能被再次配置。当再使用一个编解码器 (Codec) 时, 必须调用 `release` 函数, 不然会出现资源未被释放的问题。极少数情况下的 MediaCodec 可能会遇到一个 Error (错误) 状态。可以调用 `reset` 函数让编解码器 (Codec) 再次可用。

5. 特定编解码格式数据

特定编解码格式数据, 我们暂且将其理解为一些重要编解码信息的载体。在 MediaCodec 编解码过程中, 当遇到 AAC 音频格式及 MPEG-4、H.264 和 H.265 视频格式时, 需要设置大量的 Buffer, 或者将特定信息载体作为真实数据的开头, 如 H.264 中就包括 SPS、PPS 等重要信息 (在 H.264 相关章节中会详细介绍)。在处理这样的压缩格式, 进行编解码操作前, 调用 `start` 函数后, 这些载体是编解码的依据, 所以要立刻传给 MediaCodec。当调用 MediaCodec 的 `queueInputBuffer` 函数时, 这些数据必须使用 flag: `BUFFER_FLAG_CODEC_CONFIG`, 表明已

Android 音视频开发

经配置好，可以进行真正的编解码操作了。

特定编解码格式数据也可以包含在 `ByteBuffer` 条目的格式中传递给配置钥匙“`csd-0`”、“`csd-1`”等。这些钥匙总是包含从 `MediaExtractor` 类中获取视频源的格式。当调用 `start` 函数时，特定编解码格式数据格式自动提交给编解码器，如果不包含特定编码的数据格式，你可以按照格式要求选择以正确的顺序提交到指定的缓冲区。对于 H.264 (AVC)，还可以连接所有信息载体数据并提交作为单个编解码配置缓冲区。

Android 使用如图 6-7 所示的特定编解码格式数据缓冲区。这些也需要跟踪格式中设置适当的 `MediaMuxer Track` 配置。每个参数设置和特定编解码格式数据部分标注 (*) 必须以“`\x00\x00\x00\x01`”开始。

必须注意在执行 `start` 函数之后需要马上刷新编解码器，在任何输出 `Buffer` 或输出格式被改变并返回之前，特定编解码格式数据也许会丢失在刷新过程中。在这种刷新中，要确保合适的编解码器做编解码操作不出问题，必须重新提交使用带 `BUFFER_FLAG_CODEC_CONFIG` 标志的 `Buffer` 数据。

Format	CSD buffer #0	CSD buffer #1	CSD buffer #2
AAC	Decoder-specific information from ESDS*	Not Used	Not Used
VORBIS	Identification header	Setup header	Not Used
OPUS	Identification header	Pre-skip in nanosecs (unsigned 64-bit native-order integer.) This overrides the pre-skip value in the identification header.	Seek Pre-roll in nanosecs (unsigned 64-bit native-order integer.)
MPEG-4	Decoder-specific information from ESDS*	Not Used	Not Used
H.264 AVC	SPS (Sequence Parameter Sets*)	PPS (Picture Parameter Sets*)	Not Used
H.265 HEVC	VPS (Video Parameter Sets*) + SPS (Sequence Parameter Sets*) + PPS (Picture Parameter Sets*)	Not Used	Not Used
VP9	VP9 CodecPrivate Data (optional)	Not Used	Not Used

图 6-7 特定编解码格式数据缓冲区

在任意有效的输出 `Buffer` 带有 `codec-config` 标志前，如果是编码过程，编码器将创建并返回特定编码的数据；如果是解码过程，解码器将解出对应的 YUV 数据。

6. Codec 数据处理过程

每个 Codec 维护的一组输入和输出 `Buffer` 都指向 API 调用的 `bufferid`。成功调用 `start` 函数后，客户端“拥有”输入和输出 `Buffer`。在同步模式下，调用 `dequeueInput / OutputBuffer(...)` 来获得（所有权）Codec 的输入或输出 `Buffer`。在异步模式下，通过 `MediaCodec.Callback`。

onInput/OutputBufferAvailable(...)回调函数将自动接收回调后的 Buffer。

Codec 在异步模式下通过 onOutputBufferAvailable 回调函数将返回一个只读输出 Buffer，或在同步模式下响应 dequeueOutputBuffer 调用。在输出 Buffer 被处理后，调用 releaseOutputBuffer 函数返回 Codec 的 Buffer。

根据 API 版本，可以在 3 个方面处理数据，如图 6-8 所示。

Processing Mode	API version <= 20 Jelly Bean/KitKat	API version >= 21 Lollipop and later
Synchronous API using buffer arrays	Supported	Deprecated
Synchronous API using buffers	Not Available	Supported
Asynchronous API using buffers	Not Available	Supported

图 6-8 根据 API 版本在 3 方面处理数据

(1) 异步处理使用缓冲区

自 Android 5.0 版本开始，首选异步模式处理数据，处理方法是在调用 configure 函数之前设置一个回调。异步模式改变了状态转换，因为刷新后必须调用 start 函数，MediaCodec 过渡到 Running 子状态，这时开始接收输入 Buffer。同样，在初始直接调用 start 函数后，MediaCodec 进入 Running 状态，通过回调传递可用的输入 Buffer，如图 6-9 所示。

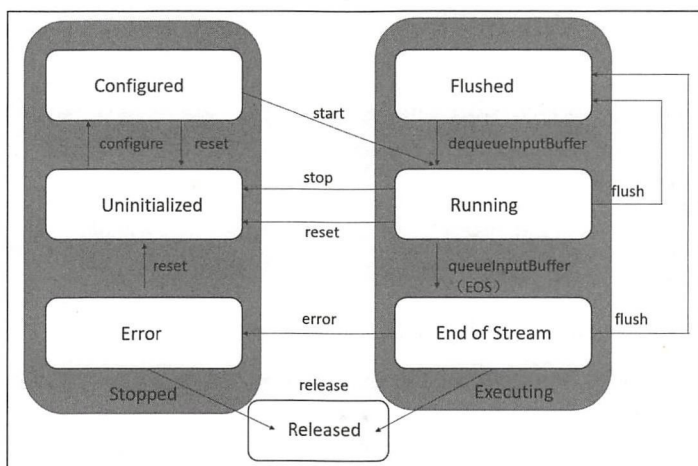


图 6-9 异步处理使用缓冲区

(2) 同步解码和异步解码

同步方式：

错误使用的原因是，开发者会从主观上认为，MediaCodec 的工作方式是提供一个 input 数

Android 音视频开发

据就输出一个对应的 output 数据。事实上完全不是这样的，MediaCodec 在读入 input 数据之后，按照自己的逻辑解码（编码），可能拆分解码（编码）数据，或者合并输出。MediaCodec 的使用者应该认为 input 和 output 没有对应关系，使用者的角色是完全被动的，把 Buffer 的使用权交给 MediaCodec。

异步方式：

Android 5.0 版本后开始支持异步解码，代码如下：

```
MediaCodec codec = MediaCodec.createByCodecType(mimeType);
codec.setCallback(new MediaCodec.Callback() {
    void onInputBufferAvailable(MediaCodec mc, int inputBufferId) {
        ByteBuffer inputBuffer = codec.getInputBuffer(inputBufferId);
        //把有效数据填入 inputBuffer 队列中
        codec.queueInputBuffer(inputBufferId, ...);
    }
    void onOutputBufferAvailable(MediaCodec mc, int outputBufferId, ...) {
        ByteBuffer outputBuffer = codec.getOutputBuffer(outputBufferId);
        MediaFormat bufferFormat = codec.getOutputFormat(outputBufferId);
        ...
        codec.releaseOutputBuffer(outputBufferId, ...);
    }
    void onError(...) {
        ...
    }
});
codec.configure(format, ...);
mOutputFormat = codec.getOutputFormat();
codec.start();
//等待处理完成
codec.stop();
codec.release();
```

查看一个案例，用 MediaCodec 来编解码一个视频到 SurfaceView 显示，代码如下：

```
public class MainActivity extends AppCompatActivity implements
SurfaceHolder.Callback {
    private static final String SAMPLE = Environment.
getExternalStorageDirectory() + "/device-2016-11-15.mp4";
    private static final String TAG = MainActivity.class.getSimpleName();
    private WorkThread mWorkThread = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```


第6章 OpenMAX (OMX) 框架

```

        SurfaceView surfaceView = new SurfaceView(this);
        /*下面设置 Surface 不维护自己的缓冲区, 而是等待屏幕的渲染引擎将内容推送到用
        户面前*/

        surfaceView.getHolder().setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
        surfaceView.getHolder().addCallback(this);
        setContentView(surfaceView);
    }

    protected void onDestroy() {
        super.onDestroy();
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int
width, int height) {
        if (mWorkThread == null) {
            mWorkThread = new WorkThread(holder.getSurface());
            mWorkThread.start();
        }
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        if (mWorkThread != null) {
            mWorkThread.interrupt();
        }
    }

    private class WorkThread extends Thread {
        private MediaExtractor mMediaExtractor;
        private MediaCodec mMediaCodec;
        private Surface mSurface;

        public WorkThread(Surface surface) {
            this.mSurface = surface;
        }

        @Override
        public void run() {

```


Android 音视频开发

```

mMediaExtractor = new MediaExtractor();//数据解析器
try {
    mMediaExtractor.setDataSource(SAMPLE);
} catch (IOException e) {
    e.printStackTrace();
}

for (int i = 0; i < mMediaExtractor.getTrackCount(); i++) {
    //遍历数据源音视频轨迹
    MediaFormat format = mMediaExtractor.getTrackFormat(i);
    Log.d(TAG, ">> format i " + i + ": " + format);
    String mime = format.getString(MediaFormat.KEY_MIME);
    Log.d(TAG, ">> mime i " + i + ": " + mime);
    if (mime.startsWith("video/")) {
        mMediaExtractor.selectTrack(i);
        try {
            mMediaCodec = MediaCodec.createDecoderByType (mime);
        } catch (IOException e) {
            e.printStackTrace();
        }
        mMediaCodec.configure(format, mSurface, null, 0);
        break;
    }
}
if (mMediaCodec == null) {
    return;
}
mMediaCodec.start();
//调用 start 函数后, 如果没有异常信息, 就表示成功构建了组件
ByteBuffer[] inputBuffers = mMediaCodec.getInputBuffers();
ByteBuffer[] outputBuffers = mMediaCodec.getOutputBuffers();
//每个 Buffer 的元数据包括具体范围偏移及大小, 以及有效数据中相关解码的 Buffer
MediaCodec.BufferInfo info = new MediaCodec.BufferInfo();
boolean isEOS = false;
long startMs = System.currentTimeMillis();
while (!Thread.interrupted()) { //只要线程不中断
    if (!isEOS) {
        //返回使用有效输出的 Buffer 的索引, 如果没有相关 Buffer 可用, 就返回-1; 如果传入的 timeoutUs 为 0, 将立马返回; 如果输入 Buffer 可用, 将无限期等待 timeoutUs 的单位是 μs
        int inIndex = mMediaCodec.dequeueInputBuffer (10000);
        //0.01s
        if (inIndex >= 0) {
            ByteBuffer buffer = inputBuffers[inIndex];

```

第6章 OpenMAX (OMX) 框架

```

        Log.d(TAG, ">> buffer " + buffer);
        int sampleSize = mMediaExtractor.
            readSampleData(buffer, 0);
        Log.d(TAG, ">> sampleSize " + sampleSize);
        if (sampleSize < 0) {
            Log.d(TAG, "InputBuffer BUFFER_FLAG_END_OF_
                STREAM");
            mMediaCodec.queueInputBuffer(inIndex, 0, 0, 0,
                MediaCodec.BUFFER_FLAG_END_OF_STREAM);
            isEOS = true;
        } else {
            mMediaCodec.queueInputBuffer(inIndex, 0,
sampleSize, mMediaExtractor.getSampleTime(), 0);
            mMediaExtractor.advance();
        }
    }
}

int outIndex = mMediaCodec.dequeueOutputBuffer(info, 10000);
switch (outIndex) {
    case MediaCodec.INFO_OUTPUT_BUFFERS_CHANGED:
        //当 Buffer 变化时, 客户端必须重新指向新的 Buffer
        Log.d(TAG, ">> output buffer changed ");
        outputBuffers = mMediaCodec.getOutputBuffers();
        break;
    case MediaCodec.INFO_OUTPUT_FORMAT_CHANGED:
        //当 Buffer 的封装格式变化时, 须指向新的 Buffer 格式
        Log.d(TAG, ">> output buffer changed ");
        break;
    case MediaCodec.INFO_TRY_AGAIN_LATER:
        //当 dequeueOutputBuffer 超时, 会到达此 case
        Log.d(TAG, ">> dequeueOutputBuffer timeout ");
        break;
    default:
        ByteBuffer buffer = outputBuffers[outIndex];
        //这里使用简单时钟的方式保持视频的 fps (每秒显示帧数), 不然视频会播放得很快
        while (info.presentationTimeUs / 1000 > System.
currentTimeMillis() - startMs) {
            try {
                sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
                break;
            }
        }
    }
}

```


Android 音视频开发

```

        }
        mMediaCodec.releaseOutputBuffer(outIndex, true);
        break;
    }

    //在所有解码后的帧被渲染后，就可以停止播放了
    if ((info.flags & MediaCodec.BUFFER_FLAG_END_OF_STREAM) !=
0) {
        Log.d(TAG, "OutputBuffer BUFFER_FLAG_END_OF_STREAM");
        break;
    }
}
mMediaCodec.stop();
mMediaCodec.release();//释放组件
mMediaExtractor.release();
}
}
}

```

案例工程代码已上传到我的 [GitHub](#)，之前其他多媒体框架相关案例的代码也在此仓库中。

6.3.2 从创建到 Start 过程

前面介绍了 `MediaCodec` 的说明及状态图，本节将深入源码中看看其过程，将主要介绍如下内容。

- `MediaCodec` 从创建到 start 过程（到 JNI 部分）。
- 补充 `MediaCodec` 基本用法。
- `MediaCodec` 中的 `BufferInfo` 内部类。

1. `MediaCodec` 从创建到 start 过程（到 JNI 部分）

通常我们用得较多的是 `MediaCodec.java` 类，从创建 `MediaCodec` 到调用 `start` 函数，其中需要历经 JNI 层相关操作，时序图如图 6-10 所示。

2. 补充 `MediaCodec` 基本用法

使用 `MediaCodec` 遵循一个基本模式，介绍如下。

（1）创建和配置 `MediaCodec` 对象。

（2）进行以下循环：如果一个输入缓冲区准备好，读取部分数据，复制到缓冲区；如果一个输出缓冲区准备好，复制到缓冲区。

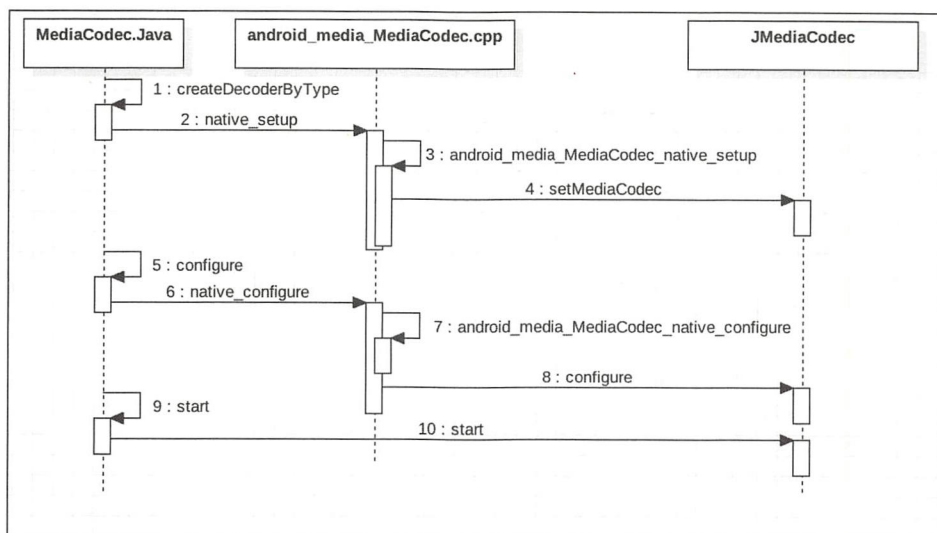


图 6-10 MediaCodec 从创建到 start 过程时序图

(3) 销毁 MediaCodec 对象。

一个 MediaCodec 对象可以对特定类型的数据 (MP3 音频或 H.264 视频) 进行编码或解码。因为在原始数据上操作, 所以任何文件头 (比如 ID3 tags) 必须被剔除, MediaCodec 不与任何更高层次的内容交互, 所以无法通过扬声器播放音频或者从网络接收视频流。它只读入缓冲区数据, 再输出到缓冲区。MediaCodec 可以把大部分的外层数据去掉。

必须将 MediaCodec 的输入处理成特定的格式。H.264 视频编码输入的就是 1 帧数据, H.264 解码指的是一个 NAL 单元。但你不可能一次只提交单个数据或数据在需要处理的时候才出现, 这样看起来, 输入更像是一个流。实际上, 编解码器在输出前同时拥有多个 Buffer。

3. MediaCodec 中的 BufferInfo 内部类

下面先看看 BufferInfo 内部类的代码:

```

public final static class BufferInfo {
    //更新 Buffer 元数据
    //newOffset 表示起始偏移量, newSize 表示缓冲区的数据量, newTimeUs 表示以 μs 为
    //单位的时间戳, newFlags 表示缓冲区标记, 这个标记在 BUFFER_FLAG_KEY_FRAME 和
    //BUFFER_FLAG_END_OF_STREAM 之间选择
    public void set(int newOffset, int newSize, long newTimeUs,
        @BufferFlag int newFlags) {
        offset = newOffset;
        size = newSize;
        presentationTimeUs = newTimeUs;
    }
}
  
```



```
        flags = newFlags;
    }

    //Buffer 中数据的起始偏移量
    public int offset;
    public int size;
    public long presentationTimeUs;
    //编码的 Buffer 中的关键帧带有 BUFFER_FLAG_KEY_FRAME 标记, 最后输出
    //Buffer 带有 BUFFER_FLAG_END_OF_STREAM 标记。在某些情况下, 这可能是一个空缓冲
    //区, 其唯一的作用是标记 End-of-Stream
    @BufferFlag
    public int flags;
    @NonNull
    public BufferInfo dup() {
        BufferInfo copy = new BufferInfo();
        copy.set(offset, size, presentationTimeUs, flags);
        return copy;
    }
};
//标记编解码数据中包含的关键帧
public static final int BUFFER_FLAG_KEY_FRAME = 1;
//标记包含初始化编解码器/特定数据而不包含媒体数据
public static final int BUFFER_FLAG_CODEC_CONFIG = 2;
//表示信号流的末尾, 之后没有可用的 Buffer, 当然如果调用 flush 函数, 将继续刷新 Buffer
public static final int BUFFER_FLAG_END_OF_STREAM = 4;
```

MediaCodec 和 MediaPlayer 在很多地方有相似之处, 当 Java 层调用 MediaCodec.createByCodecName、MediaCodec.createDecoderByType、MediaCodec.createEncoderByType 时, 都会执行 MediaCodec 的构造函数, 构造函数中都会调用 native_setup, 代码如下:

```
private MediaCodec(String name, boolean nameIsType, boolean encoder) {
    native_setup(name, nameIsType, encoder);
}
private native final void native_setup(String name, boolean nameIsType,
boolean encoder);
```

其中对应有这么一段代码, 相当于做了一次映射:

```
static JNINativeMethod gMethods[] = {
    { "native_release", "()V", (void *)android_media_MediaCodec_
release },
    { "native_reset", "()V", (void *)android_media_MediaCodec_reset },
    //省略部分代码
    { "native_setup", "(Ljava/lang/String;ZZ)V",
    (void *)android_media_MediaCodec_native_setup },
```



```
{ "native_finalize", "()V",  
  (void *)android_media_MediaCodec_native_finalize },  
};
```

接着进入 `android_media_MediaCodec_native_setup` 函数，主要用于构建 JNI 层中的 `MediaCodec`，也就是 `JMediaCodec`：

```
static void android_media_MediaCodec_native_setup(  
    JNIEnv *env, jobject thiz,  
    jstring name, jboolean nameIsType, jboolean encoder) {  
    const char *tmp = env->GetStringUTFChars(name, NULL);  
    if (tmp == NULL) {  
        return;  
    }  
  
    sp<JMediaCodec> codec = new JMediaCodec(env, thiz, tmp, nameIsType,  
encoder);  
    const status_t err = codec->initCheck();  
    //省略部分代码  
    env->ReleaseStringUTFChars(name, tmp);  
    codec->registerSelf();  
    setMediaCodec(env, thiz, codec);  
}
```

创建 `JMediaCodec` 后，通过 `setMediaCodec` 设置 `JMediaCodec`，代码如下：

```
static sp<JMediaCodec> setMediaCodec(  
    JNIEnv *env, jobject thiz, const sp<JMediaCodec> &codec) {  
    sp<JMediaCodec> old = (JMediaCodec *)env->GetLongField(thiz, gFields.  
context);  
    if (codec != NULL) {  
        codec->incStrong(thiz);  
    }  
    if (old != NULL) {  
        old->release();  
        old->decStrong(thiz);  
    }  
    env->SetLongField(thiz, gFields.context, (jlong)codec.get());  
    return old;  
}
```

上面用到了智能指针中的成员函数 `incStrong` 和 `decStrong` 来维护引用计数器的值，这两个函数就是提供给智能指针来调用的。要注意的是，在 `decStrong` 函数中，如果当前引用计数值为 1，那么当减 1 后就会变成 0，于是就会删除这个对象。接下来看看 `JMediaCodec` 的构造函数，代码如下：



```
JMediaCodec::JMediaCodec(
    JNIEnv *env, jobject thiz,
    const char *name, bool nameIsType, bool encoder)
: mClass(NULL),
  mObject(NULL) {
    jclass clazz = env->GetObjectClass(thiz);
    CHECK(clazz != NULL);

    mClass = (jclass)env->NewGlobalRef(clazz);
    mObject = env->NewWeakGlobalRef(thiz);
    cacheJavaObjects(env);
    mLooper = new ALooper();
    mLooper->setName("MediaCodec_looper");
    mLooper->start(
        false,          //runOnCallingThread
        true,           //canCallJava
        PRIORITY_FOREGROUND);

    if (nameIsType) {
        mCodec = MediaCodec::CreateByType(mLooper, name, encoder,
&mInitStatus);
    } else {
        mCodec = MediaCodec::CreateByComponentName(mLooper, name,
&mInitStatus);
    }
    CHECK((mCodec != NULL) != (mInitStatus != OK));
}
```

从构造函数中可以看到，JMediaCodec 最终执行到 C++层 MediaCodec 的两个 Create 函数，通过以上几个步骤得到 MediaCodec 对象后，就开始执行 Java 层调用 MediaCodec.configure 函数，代码如下：

```
public void configure(
    MediaFormat format,
    Surface surface, MediaCrypto crypto, int flags) {
    Map<String, Object> formatMap = format.getMap();
    String[] keys = null;
    Object[] values = null;
    if (format != null) {
        keys = new String[formatMap.size()];
        values = new Object[formatMap.size()];
        int i = 0;
        for(Map.Entry<String, Object> entry: formatMap.entrySet()) {
            keys[i] = entry.getKey();
            values[i] = entry.getValue();
        }
    }
}
```



```
        ++i;
    }
}
native_configure(keys, values, surface, crypto, flags);
}
```

获取 format 中的 Map，实际上是一个 HashMap，然后遍历视频源的格式名并存放到两个数组中，再通过 android_media_MediaCodec_native_configure 向下传递，代码如下：

```
static void android_media_MediaCodec_native_configure(JNIEnv *env, jobject
thiz, jobjectArray keys, jobjectArray values, jobject jsurface, jobject jcrypto,
jint flags) {
    sp<JMediaCodec> codec = getMediaCodec(env, thiz);
    sp<AMessage> format;
    status_t err = ConvertKeyValueArraysToMessage(env, keys, values,
&format);
    //省略部分代码
    sp<IGraphicBufferProducer> bufferProducer;
    if (jsurface != NULL) {
        sp<Surface> surface(android_view_Surface_getSurface(env, jsurface));
        if (surface != NULL) {
            bufferProducer = surface->getIGraphicBufferProducer();
            //Surface 就相当于一块 Buffer，调用 getIGraphicBufferProducer
            //获取对应的 IGraphicBufferProducer
        }
        //省略部分代码
    }

    sp<ICrypto> crypto;
    if (jcrypto != NULL) {
        crypto = JCrypto::GetCrypto(env, jcrypto);
    }
    err = codec->configure(format, bufferProducer, crypto, flags);
}
```

通过 getMediaCodec 获取对应的 JMediaCodec，接着传入 jsurface 变量，表示 JNI 层中的 Surface 数据，这个 jsurface 变量不一定是真正的 Surface（Surface 实际上是一块 Buffer），也有可能是 SurfaceHolder（SurfaceHolder 可以通过其中的 getSurface 获取 Surface），最后调用 JMediaCodec 的 configure 函数，代码如下：

```
status_t JMediaCodec::configure(
    const sp<AMessage> &format,
    const sp<IGraphicBufferProducer> &bufferProducer,
    const sp<ICrypto> &crypto,
    int flags) {
```




```
sp<Surface> client;
if (bufferProducer != NULL) {
    mSurfaceTextureClient =
        new Surface(bufferProducer, true /* controlledByApp */);
} else {
    mSurfaceTextureClient.clear();
}
return mCodec->configure(format, mSurfaceTextureClient, crypto, flags);
}
```

上面最后的 `mCodec->configure` 最终会调用 `MediaCodec.cpp` 中的 `configure` 函数，构建一些编解码器。在 `MediaCodec.java` 调用 `start` 函数后，会执行如下代码：

```
static void android_media_MediaCodec_start(JNIEnv *env, jobject thiz) {
    ALOGV("android_media_MediaCodec_start");
    sp<JMediaCodec> codec = getMediaCodec(env, thiz);
    //省略部分代码
    status_t err = codec->start();
    throwExceptionAsNecessary(env, err, ACTION_CODE_FATAL, "start failed");
}
```

在上面的代码获取 `JMediaCodec` 后，调用 `JMediaCodec` 中的 `start` 函数，`JMediaCodec` 中的 `start` 函数代码如下：

```
status_t JMediaCodec::start() {
    return mCodec->start();
}
```

这个 `mCodec` 最后会执行 C++ 的 `MediaCodec.cpp` 中的 `start` 函数。最后到达 `ACodec.cpp` 的 `start` 函数中去执行解码操作。

6.3.3 MediaCodec 到 OMX 框架过程

在介绍 `NuPlayer` 时，`NuPlayer` 解码部分会创建 `MediaCodec`，并且最终到达 `OMX` 框架，先看看 `MediaCodec` 的 `init` 函数：

```
status_t MediaCodec::init(const AString &name, bool nameIsType, bool
encoder) {
    mResourceManagerService->init();
    if (nameIsType || !strncasecmp(name.c_str(), "omx.", 4)) {
        mCodec = new ACodec;
    } else if (!nameIsType
        && !strncasecmp(name.c_str(), "android.filter.", 15)) {
        mCodec = new MediaFilter;
    } else {
```




```
        return NAME_NOT_FOUND;
    }

    bool secureCodec = false;
    if (nameIsType && !strncasecmp(name.c_str(), "video/", 6)) {
        mIsVideo = true;
    } else {
        //省略部分代码
    }

    if (mIsVideo) {
        //视频编解码器需要专用的 Looper (消息轮循器)
        if (mCodecLooper == NULL) {
            mCodecLooper = new ALooper;
            mCodecLooper->setName("CodecLooper");
            mCodecLooper->start(false, false, ANDROID_PRIORITY_AUDIO);
        }
        mCodecLooper->registerHandler(mCodec);
    } else {
        mLooper->registerHandler(mCodec);
    }
    mLooper->registerHandler(this);
    mCodec->setNotificationMessage(new AMessage(kWhatCodecNotify, this));
    sp<AMessage> msg = new AMessage(kWhatInit, this);
    //省略部分代码
    return err;
}
```

从 init 函数中可以看到, 首先创建了 ACodec, 并且初始化了 ALooper、AMessage, 由于 ACodec 继承自 AHandler, 那么一套消息机制就有了。最后发送 kWhatInit 消息, 收到消息的逻辑位于 frameworks/av/media/libstagefright/ACodec.cpp 中, 代码如下:

```
case kWhatInit:
{
    //省略部分代码
    mCodec->initiateAllocateComponent(format);
    break;
}
```

主要是调用了 ACodec 的 initiateAllocateComponent 函数:

```
void ACodec::initiateAllocateComponent(const sp<AMessage> &msg) {
    msg->setWhat(kWhatAllocateComponent);
    msg->setTarget(this);
    msg->post();
}
```



```
}
```

同样是发送了一个 `kWhatAllocateComponent` 消息，消息中心收到后，会调用 `onAllocateComponent` 回调函数，代码如下：

```
bool ACodec::UninitializedState::onAllocateComponent(const sp<AMessage>
&msg) {
    //省略部分代码
    OMXClient client;
    if (client.connect() != OK) {
        mCodec->signalError(OMX_ErrorUndefined, NO_INIT);
        return false;
    }
    sp<IOMX> omx = client.interface();//通过 OMXClient 获取 IOMX
    Vector<OMXCodec::CodecNameAndQuirks> matchingCodecs;
    AString mime;
    AString componentName;
    uint32_t quirks = 0;
    int32_t encoder = false;
    if (msg->findString("componentName", &componentName)) {
        ssize_t index = matchingCodecs.add();
        OMXCodec::CodecNameAndQuirks *entry = &matchingCodecs.
editItemAt(index);
        entry->mName = String8(componentName.c_str());
        if (!OMXCodec::findCodecQuirks(
            componentName.c_str(), &entry->mQuirks)) {
            entry->mQuirks = 0;
        }
    } else {
        OMXCodec::findMatchingCodecs(
            mime.c_str(),
            encoder, //createEncoder
            NULL,    //matchComponentName
            0,       //flags
            &matchingCodecs);
    }

    sp<CodecObserver> observer = new CodecObserver;
    IOMX::node_id node = 0;
    status_t err = NAME_NOT_FOUND;
    for (size_t matchIndex = 0; matchIndex < matchingCodecs.size();
        ++matchIndex) {
        componentName = matchingCodecs.itemAt(matchIndex).mName.string();
        quirks = matchingCodecs.itemAt(matchIndex).mQuirks;
    }
}
```




```
pid_t tid = gettid();
int prevPriority = androidGetThreadPriority(tid);
androidSetThreadPriority(tid, ANDROID_PRIORITY_FOREGROUND);
//分配 Node 节点
err = omx->allocateNode(componentName.c_str(), observer, &node);
androidSetThreadPriority(tid, prevPriority);
node = 0;
}
//省略部分代码
mCodec->mQuirks = quirks;
mCodec->mOMX = omx;
mCodec->mNode = node;
//省略部分代码
return true;
}
```

从这里可以看到, 通过 ACodec 中的 AllocateComponent 函数, 主要是先判断 OMXClient 和 Server 是否正常地建立了连接, 然后通过 IOMX 进行 IPC 通信, 接着调用 omx->allocateNode 分配 Node 节点。

同理, 看看 onConfigureComponent 函数:

```
bool ACodec::LoadedState::onConfigureComponent(const sp<AMessage> &msg)
{
    status_t err = OK;
    AString mime;
    if (!msg->findString("mime", &mime)) {
        err = BAD_VALUE;
    } else {
        err = mCodec->configureCodec(mime.c_str(), msg);
        //这里的 mCodec 是 ACodec
    }
    return true;
}
```

上面的代码会调用 ACodec 的 configureCodec 函数, 代码如下:

```
status_t ACodec::configureCodec(
    const char *mime, const sp<AMessage> &msg) {
    //省略部分代码
    if (encoder) {
        err = setupVideoEncoder(mime, msg);
    } else {
        err = setupVideoDecoder(mime, msg, haveNativeWindow);
    }
}
```




```
err = initNativeWindow(); //初始化 NativeWindow
err = setupAACCodec(encoder, numChannels, sampleRate, bitRate,
aacProfile, isADTS != 0, sbrMode, maxOutputChannelCount, drc, pcmLimiterEnable);
err = setupG711Codec(encoder, sampleRate, numChannels);
//省略部分代码
err = setupFlacCodec(encoder, numChannels, sampleRate, compressionLevel);
//省略部分代码
err = setupRawAudioFormat(kPortIndexInput, sampleRate, numChannels);
//省略部分代码
err = setupAC3Codec(encoder, numChannels, sampleRate);
//省略部分代码
err = setupEAC3Codec(encoder, numChannels, sampleRate);
//省略部分代码
int32_t maxInputSize;
if (msg->findInt32("max-input-size", &maxInputSize)) {
    err = setMinBufferSize(kPortIndexInput, (size_t)maxInputSize);
} else if (!strcmp("OMX.Nvidia.aac.decoder", mComponentName.c_str()))
{
    err = setMinBufferSize(kPortIndexInput, 8192); // XXX
}
//省略部分代码
}
```

ConfigureCode 主要用于构建一些编解码器，包括各种不同的音频编解码器和视频编解码器。

6.3.4 MediaCodec 硬解码

MediaCodec 调用的是在系统中注册过的解码器，硬件厂商会把自己的硬解码器注册进来，这就是硬解码，如果厂商注册一个软解码器，则是软解码。

MediaCodec 并不是真正的编解码器，真正的编解码器是在 OpenMAX 中，要保证是硬解码，在 MediaCodec 里有接口可以枚举所有解码器，每种编码可能都有多个解码器，区分哪个是软解码哪个是硬解码就行了。如通过 mime 构建 MediaCodec，代码如下：

```
MediaCodec mediaCodec = MediaCodec.createDecoderByType("video/avc");
```

我的应用里面接收的是 H.264 编码数据，所以我选取的是 video/avc，可以看一下 MediaCodec.createDecoderByType()枚举了哪些编解码器：

```
/**
 * 实例化一个支持给定 mime 类型的输入数据的解码器
 * 以下是已定义 mime 类型及其语义的列表
 * <ul>
```



```
* <li>"video/x-vnd.on2.vp8" - VP8 video (i.e. video in .webm)
* <li>"video/x-vnd.on2.vp9" - VP9 video (i.e. video in .webm)
* <li>"video/avc" - H.264/AVC video
* <li>"video/mp4v-es" - MPEG4 video
* <li>"video/3gpp" - H.263 video
* <li>"audio/3gpp" - AMR narrowband audio
* <li>"audio/amr-wb" - AMR wideband audio
* <li>"audio/mpeg" - MPEG1/2 audio layer III
* <li>"audio/mp4a-latm" - AAC audio (note, this is raw AAC packets,
not packaged in LATM!)
* <li>"audio/vorbis" - vorbis audio
* <li>"audio/g711-alaw" - G.711 alaw audio
* <li>"audio/g711-mlaw" - G.711 ulaw audio
* </ul>
*
* @param type The mime type of the input data.
*/
public static MediaCodec createDecoderByType(String type) {
    return new MediaCodec(type, true /* nameIsType */, false /*
encoder */);
}
```

可以看到我选的“video/avc” - H.264/AVC video 是一种 H.264 解码方式，但并不能证明我使用的就一定是硬解码。

下面先来看一下 Android 系统中解码器的命名方式，软解码器通常是以 OMX.google 开头的，硬解码器通常是以 OMX.[hardware_vendor]开头的，比如 MTK 的解码器是以 OMX.MTK 开头的。当然还有一些不遵守这个命名规范的，不以 OMX.开头的情况，它们也会被认为是软解码器。

判断规则见 frameworks/av/media/libstagefright/OMXCodec.cpp:

```
static bool IsSoftwareCodec(const char *componentName) {
    if (!strncmp("OMX.google.", componentName, 11)) {
        return true;
    }
    if (!strncmp("OMX.", componentName, 4)) {
        return false;
    }
    return true;
}
```

其实 MediaCodec 调用的是系统中注册的解码器，系统中可以存在很多解码器，但能够被应用的解码器是根据配置来的，即/system/etc/media_codecs.xml。这个文件一般由硬件或者系



统的生产厂家在编译整个系统的时候提供，一般保存在代码的 `device/[company]/[codename]` 目录下。这个文件配置了系统中有哪些可用的 Codec 以及这些 Codec 对应的媒体文件类型。在这个文件里面，系统提供的软硬编解码器都需要被列出来。

也就是说，如果系统里面实际上包含了某个 Codec，但是并没有被配置在这个文件里，那么应用程序也无法使用。

在这个配置文件里面，如果出现多个 Codec 对应同样类型的媒体格式，这些 Codec 都会被保留。当系统使用的时候，将会选择第一个匹配的 Codec，除非指明了要软解码还是硬解码。但是在 Android 的 framework 层为上层提供服务的 `AwesomePlayer` 处理音频和视频的时候，对到底是选择软解码还是硬解码的参数没有设置。所以虽然底层是支持选择的，但是对于上层使用 `MediaPlayer` 的 Java 程序来说，还是只能接受默认的 Codec 选取规则。

Android 提供的命令行程序 `/system/bin/stagefright` 在播放音频文件的时候，倒是可以根据参数来选择到底使用软解码还是硬解码，但是该工具只支持播放音频，不支持播放视频。一般来说，如果系统里面有对应的媒体硬件解码器，系统开发人员应该会将其配置在 `media_codecs.xml` 中，所以在大多数情况下，如果有硬件解码器，那么我们总是会用到。在极少数情况下，硬件解码器存在，但不配置，我猜只可能是这个硬件解码器还有 Bug，暂时还不适合发布，所以不使用。



第 7 章

FFmpeg 项目

FFmpeg 是广泛使用的多媒体解决方案，几乎音视频方面的开发者没有不知道它的。之所以应用这么广泛，是因为其功能强大，所以学习 FFmpeg 是音视频开发者所必需的。

7.1 FFmpeg 简介

FFmpeg 是一个自由软件项目的名称，采用 LGPL 或 GPL 许可证。它提供了录制、转换以及流化音视频的完整解决方案。它包含非常先进的音频/视频编解码库 libavcodec，为了保证高可移植性和编解码质量，libavcodec 里的很多代码都是新开发的。“FFmpeg”这个单词中的“FF”指的是“Fast Forward”，而“mpeg”指一种压缩率比较大的活动图像和声音的压缩标准。

1. 命令行工具概述 (Command Line Tools Documentation)

下面将介绍一些 FFmpeg 使用时的基本命令，是最基础的内容。你不用太熟悉 FFmpeg 里面的代码是如何写的，如我们只需要调用

```
ffmpeg -i input.avi -r 24 output.avi
```

就可以把 input.avi 中的帧率变成 24，输出的 output.avi 就是 24fps 的视频。

- ffmpeg: 包含 FFmpeg 的各种功能，如 gif、格式转换、截图、编解码等。
- ffmpeg-all: 包含 FFmpeg 工具及 FFmpeg 组件。
- ffmpegplay: 一个使用了 FFmpeg 和 SDL 库的、简单的、可移植的媒体播放器。
- ffmpegplay-all: 包含 ffmpegplay 工具及 FFmpeg 组件。

- `ffprobe`: 用来查看多媒体文件的信息。
- `ffprobe-all`: 包含 `ffprobe` 工具及 `FFmpeg` 组件。
- `ffserver`: 与流媒体服务器相关, 负责响应客户端的流媒体请求, 把流媒体数据发送给客户端。
- `ffserver-all`: 包含 `ffserver` 工具及 `FFmpeg` 组件。

2. 组件概述 (Components Documentation)

- `Utilities`: `libavutils` 提供的通用的 `features` 及工具。
- `Video scaling and pixel format converter`: 视频缩放和像素格式转换器。
- `Audio resampler`: 音频重新取样。
- `Encoders and decoders(codecs)`: 编码和解码。
- `Bitstream filters`: 码流过滤器。
- `Muxers and demuxers(formats)`: 数据合成及数据分离。
- `Protocols`: 协议。
- `Input and output devices`: 输入和输出设备。
- `Filters`: 过滤器。

3. 类库概述 (Libraries Documentation)

- `libavutil`: 包含一些公共的工具函数的使用库, 包括算术运算、字符操作等。
- `libswscale`: (原始视频格式转换) 用于视频场景比例缩放、色彩映射转换、图像颜色空间或格式转换, 如 `RGB565`、`RGB888` 等与 `YUV420` 等之间的转换。
- `libswresample`: 原始音频格式转码。
- `libavcodec`: 用于各种类型声音/图像编解码; 该库是音视频编解码核心库, 实现了市面上可见的绝大部分编解码器的功能。
- `libavformat`: 用于各种音视频封装格式的生成和解析, 包括获取解码所需信息以生成解码上下文结构和读取音视频帧等功能; 音视频的格式解析协议, 为 `libavcodec` 分析码流提供独立的音频或视频码流源。
- `libavdevice`: 硬件采集、加速、显示。操作计算机中常用的音视频捕获或输出设备有 `ALSA`、`AUDIO_BEOS`、`JACK`、`OSS`、`1394`、`VFW` 等。
- `libavfilter`: `filter` (`FileIO`、`FPS`、`DrawText`) 音视频滤波器的开发, 如宽高比、裁剪、格式化、非格式化、伸缩等。

`FFmpeg` 源码目录结构如图 7-1 所示。

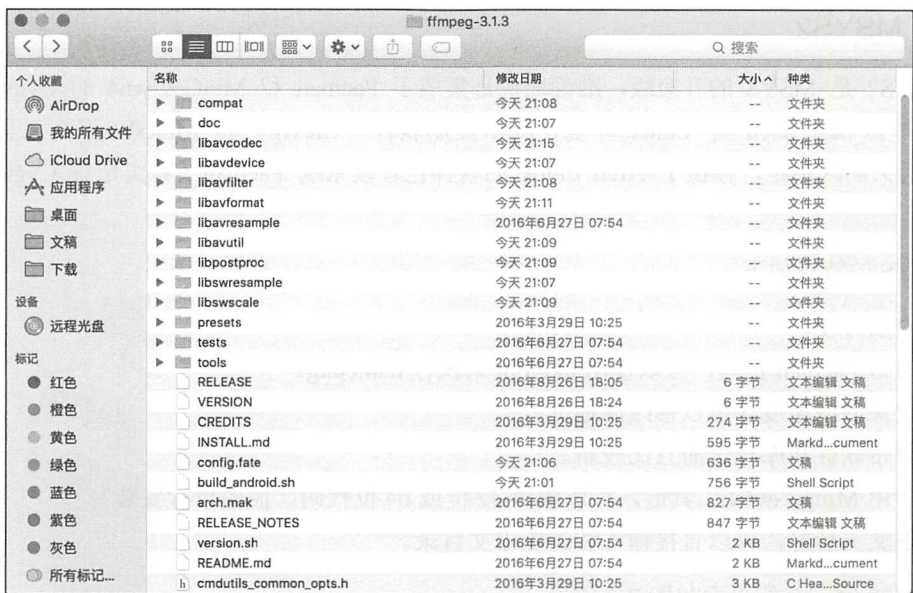


图 7-1 FFmpeg 源码目录结构

对照 FFmpeg-3.1.3，补充如下两个库。

- libavresample：音视频封装编解码格式预设等。
- libpostproc：（同步、时间计算的简单算法）用于后期效果处理，音视频应用的后处理，如图像的去块效应。

4. 文件概述（General Documentation）

- Frequently Asked Questions：常见问题。
- Supported External Libraries, Formats, Codecs or Features：支持扩展库、格式、编解码或特性。
- Platform Specific Information：平台具体信息。
- Developer Documentation：开发者概述。
- Git How-To：Git。
- FFmpeg Automated Testing Environment：FFmpeg 自动化测试环境。

7.2 在 Windows 下编译 FFmpeg

在 Windows 下编译 FFmpeg 需要准备工具 MSYS2 和 Yasm。

7.2.1 MSYS2

MSYS2 是 MSYS 的升级版，准确地说是集成了 Pacman 和 MinGW-w64 的 Cygwin 升级版，提供了 bash shell 等 Linux 环境、版本控制软件（Git/Hg）和 MinGW-w64 工具链。与 MSYS 最大的区别是，移植了 Arch Linux 的软件包管理系统 Pacman（其实是与 Cygwin 的区别）。

MSYS 的特点如下。

- 安装方便。
- 自带 Pacman 管理，可以使用 pkgtool 来执行 makepkg。
- 较快的源速度（可以修改源地址）。
- 自带软件和库较全而且比较新。
- 使用 Mingw-w64 工具链，可以编译 32 位或 64 位代码（需要自行安装）。
- 中文支持好，可以直接输入和浏览中文目录。

MSYS2 官方介绍页面如图 7-2 所示。



图 7-2 MSYS2 官方介绍页面

从官网下载计算机对应版本，进行安装就行。

7.2.2 Yasm

Yasm 是一个汇编器，一个完全重写的 NASM 汇编器。目前，它支持 X86 和 AMD64 指令集，接受 NASM 和 GNU 汇编器（GAS）语法，产出二进制、ELF32、ELF64、COFF、MachO 的（32 和 64 位）和 RDOFF2 的 Win32 和 Win64 对象的格式，并生成 STABS 调试信息的来源，即 DWARF 2、CodeView 8 格式。

下面安装 Yasm，步骤如下。

(1) 进入 Yasm 官网（如图 7-3 所示）。

(2) 单击 Download 按钮。

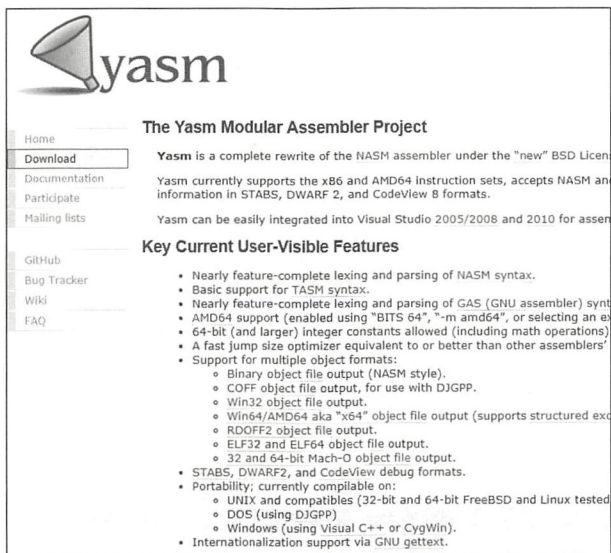


图 7-3 Yasm 官网下载页面

接下来找到和自己机器相关的版本，如我的是 X64 的计算机，则单击 Win64.exe 链接，如图 7-4 所示。

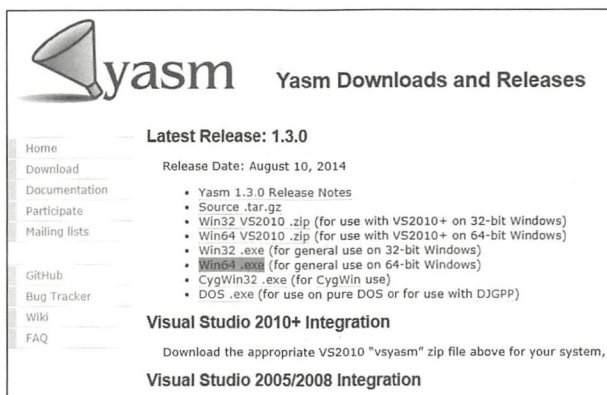


图 7-4 不同操作系统对应的 Yasm 的下载

将下载的文件直接复制到 System32 文件夹下，重命名为 yasm.exe。当然你也可以放到

MinGw/bin 下，二者并无差异，如图 7-5 所示。

名称	修改日期	类型	大小
xinput1_1.dll	2006/3/31 12:39	应用程序扩展	82 KB
xinput1_2.dll	2006/7/28 9:31	应用程序扩展	82 KB
xinput1_3.dll	2007/4/4 18:54	应用程序扩展	105 KB
XInput9_1_0.dll	2009/7/14 9:42	应用程序扩展	30 KB
xmfilter.dll	2009/7/14 9:42	应用程序扩展	66 KB
xmllite.dll	2015/3/16 18:53	应用程序扩展	195 KB
xmprovi.dll	2009/7/14 9:42	应用程序扩展	22 KB
xolehlp.dll	2009/7/14 9:42	应用程序扩展	56 KB
XpsFilt.dll	2009/7/14 9:42	应用程序扩展	946 KB
XpsGdiConverter.dll	2015/3/17 10:00	应用程序扩展	511 KB
XpsPrint.dll	2015/3/17 10:00	应用程序扩展	1,643 KB
XpsRasterService.dll	2010/11/21 11:24	应用程序扩展	225 KB
xpsrchvw.exe	2009/7/14 9:40	应用程序	4,723 KB
xpsrchvw.xml	2009/6/11 4:31	XML 文档	75 KB
xpservices.dll	2010/11/21 11:24	应用程序扩展	2,938 KB
XPSHQR.dll	2009/7/14 9:42	应用程序扩展	690 KB
xpsvcs.dll	2009/7/14 9:42	应用程序扩展	1,540 KB
xwizard.dtd	2009/6/11 5:03	XML Document T...	4 KB
xwizard.exe	2009/7/14 9:40	应用程序	42 KB
xwizards.dll	2009/7/14 9:42	应用程序扩展	423 KB
xwreg.dll	2009/7/14 9:42	应用程序扩展	100 KB
xwtpdui.dll	2009/7/14 9:42	应用程序扩展	197 KB
xwtpw32.dll	2009/7/14 9:42	应用程序扩展	127 KB
YasmshsAE.dll	2014/2/27 20:02	应用程序扩展	2,113 KB
yasm.exe	2017/12/18 6:47	应用程序	610 KB
zipfldr.dll	2010/11/21 11:24	应用程序扩展	358 KB

图 7-5 复制文件到 Windows 操作系统 System32 目录下

7.2.3 开始编译 FFmpeg-3.1.3

在编译前，在源码中修改 FFmpeg 的 configure 文件。由于编译出来的动态库文件名的版本号在 .so 之后（例如 “libavcodec.so.5.100.1”），而 Android 平台不能识别这样的文件名，所以需要修改该文件名。在 configure 文件中找到下面几行代码（在 3209~3212 行）：

```
SLIBNAME_WITH_MAJOR='$(SLIBNAME).$(LIBMAJOR) '
LIB_INSTALL_EXTRA_CMD='$(RANLIB) "$(LIBDIR)/$(LIBNAME) "'
SLIB_INSTALL_NAME='$(SLIBNAME_WITH_VERSION) '
SLIB_INSTALL_LINKS='$(SLIBNAME_WITH_MAJOR)$(SLIBNAME) '
```

替换为下面的代码：

```
SLIBNAME_WITH_MAJOR='$(SLIBPREF)$(FULLNAME)-$(LIBMAJOR)$(SLIBSUF) '
LIB_INSTALL_EXTRA_CMD='$(RANLIB) "$(LIBDIR)/$(LIBNAME) "'
SLIB_INSTALL_NAME='$(SLIBNAME_WITH_MAJOR) '
SLIB_INSTALL_LINKS='$(SLIBNAME) '
```

修改后的 configure 文件内容如图 7-6 所示。

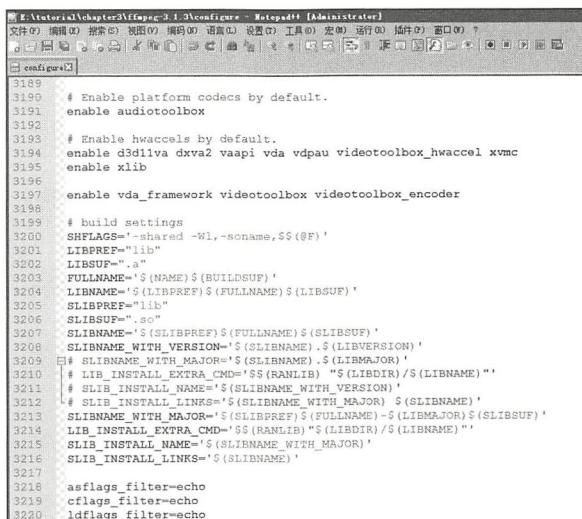


图 7-6 修改后的 configure 文件内容

7.2.4 创建 shell 编译脚本

接下来开始写 shell 脚本，进入 ffmpeg-3.1.3 目录，新建一个 build_android.sh 文件，如图 7-7 所示。

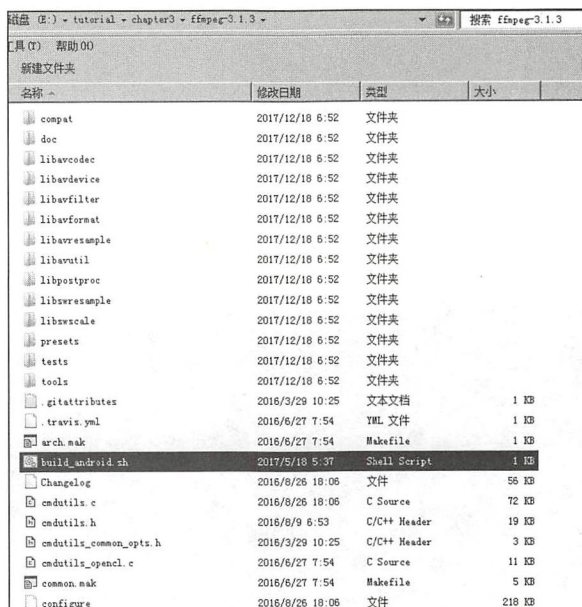
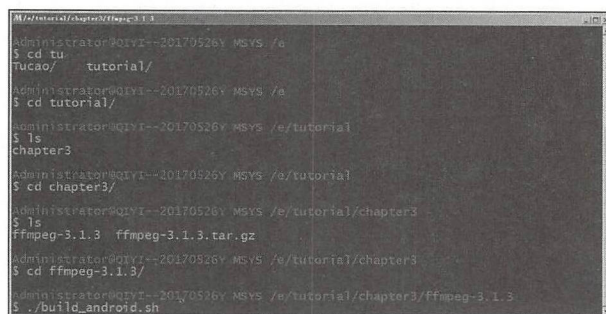


图 7-7 在 ffmpeg-3.1.3 目录中新建 build_android.sh 文件

build_android.sh 文件的内容如下:

```
#!/bin/bash
make clean
export NDK=E:/android-ndk-r9d
export SYSROOT=$NDK/platforms/android-9/arch-arm/
export TOOLCHAIN=$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/
windows-x86_64
export CPU=arm
export PREFIX=E:/tutorial/chapter3/ffmpeg-3.1.3/android/$CPU
export ADDI_CFLAGS="-marm"
./configure --target-os=android \
--prefix=$PREFIX --arch=arm \
--disable-doc \
--enable-shared \
--disable-static \
--disable-yasm \
--disable-symver \
--enable-gpl \
--disable-ffmpeg \
--disable-ffplay \
--disable-ffprobe \
--disable-ffserver \
--disable-doc \
--disable-symver \
--cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \
--enable-cross-compile \
--sysroot=$SYSROOT \
--extra-cflags="-Os -fpic $ADDI_CFLAGS" \
--extra-ldflags="$ADDI_LDFLAGS" \
$ADDITIONAL_CONFIGURE_FLAG
make clean
make
make install
```

开始执行脚本, 如图 7-8 所示。



```
Administrator@QIYI-20170526Y MSYS /e
$ cd tu-
Administrator@QIYI-20170526Y MSYS /e
$ cd tutorial/
Administrator@QIYI-20170526Y MSYS /e/tutorial
$ ls
chapter3
Administrator@QIYI-20170526Y MSYS /e/tutorial
$ cd chapter3/
Administrator@QIYI-20170526Y MSYS /e/tutorial/chapter3
$ ls
ffmpeg-3.1.3  ffmpeg-3.1.3.tar.gz
Administrator@QIYI-20170526Y MSYS /e/tutorial/chapter3
$ cd ffmpeg-3.1.3/
Administrator@QIYI-20170526Y MSYS /e/tutorial/chapter3/ffmpeg-3.1.3
$ ./build_android.sh
```

图 7-8 开始执行脚本

如果你的 MSYS2 没有更新, 最好执行 `pacman -Syu` 命令, 同步更新软件包, 如图 7-9 所示。

```

M:/tutorial/chapter3/ffmpeg-3.1.3
Administrator@QIYI--20170526Y /e/tutorial/chapter3/ffmpeg-3.1.3
$ pacman -Syu
:: 正在同步软件包数据库...
mingw32             370.1 KiB   191K/s   00:02 [#####] 100%
mingw32.sig         96.0 B     0.00B/s   00:00 [#####] 100%
mingw64             369.9 KiB   202K/s   00:02 [#####] 100%
mingw64.sig         96.0 B     93.8K/s   00:00 [#####] 100%
msys                158.9 KiB   249K/s   00:01 [#####] 100%
msys.sig            96.0 B     0.00B/s   00:00 [#####] 100%
:: Starting core system upgrade...
警告: terminate other MSYS2 programs before proceeding
正在解决依赖关系...
正在查找软件包冲突...
软件包 (5) bash-4.4.012-1 filesystem-2017.05-1 mintty-1-2.8.1-1
               msys2-runtime-2.9.0-2 pacman-5.0.1-4
下载大小:    11.52 MiB
全部安装大小:  54.80 MiB
净更新大小:    3.20 MiB
:: 进行安装吗? [Y/n] Y
:: 正在获取软件包.....
msys2-runtime-2.9.0... 2.4 MiB   340K/s   00:07 [#####] 100%

```

图 7-9 同步更新软件包过程

有时, 如果没有安装 `make`, 会提示 `make: xx:未找到信息`, 如图 7-10 所示。

```

M:/tutorial/chapter3/ffmpeg-3.1.3
Administrator@QIYI--20170526Y /e/tutorial/chapter3/ffmpeg-3.1.3
$ make clean
-bash: make: 未找到命令

Administrator@QIYI--20170526Y /e/tutorial/chapter3/ffmpeg-3.1.3
$ pacman -S make
正在解决依赖关系...
正在查找软件包冲突...
软件包 (1) make-4.2.1-1
下载大小:    0.41 MiB
全部安装大小:  1.22 MiB
:: 进行安装吗? [Y/n] Y
:: 正在获取软件包.....
make-4.2.1-1-x86_64 414.8 KiB   329K/s   00:01 [#####] 100%
(1/1) 正在检查密钥环里的密钥 [#####] 100%
(1/1) 正在检查软件包完整性 [#####] 100%
(1/1) 正在加载软件包文件 [#####] 100%
(1/1) 正在检查文件冲突 [#####] 100%
(1/1) 正在检查可用硬盘空间 [#####] 100%
:: 正在处理软件包的变化... [#####] 100%
(1/1) 正在安装 make [#####] 100%
Administrator@QIYI--20170526Y /e/tutorial/chapter3/ffmpeg-3.1.3
$

```

图 7-10 同步更新软件包完成

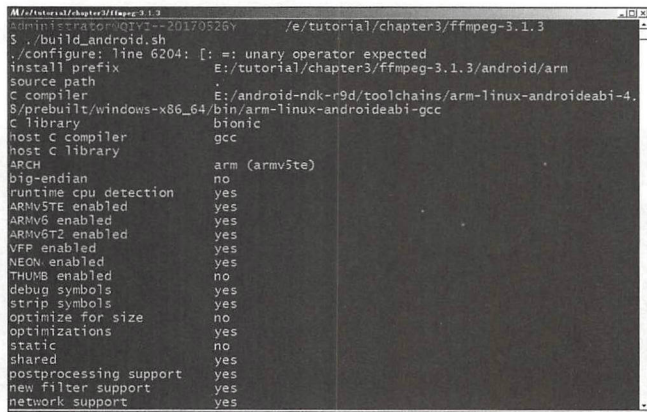
这时通过 Pacman 安装 `make` 就可以了。

7.2.5 编译动态库.so

当看到如图 7-11 所示的界面时, 说明已经开始编译 FFmpeg 了。

编译 FFmpeg 中间过程, 如图 7-12 所示。

编译 FFmpeg 完成, 如图 7-13 所示。

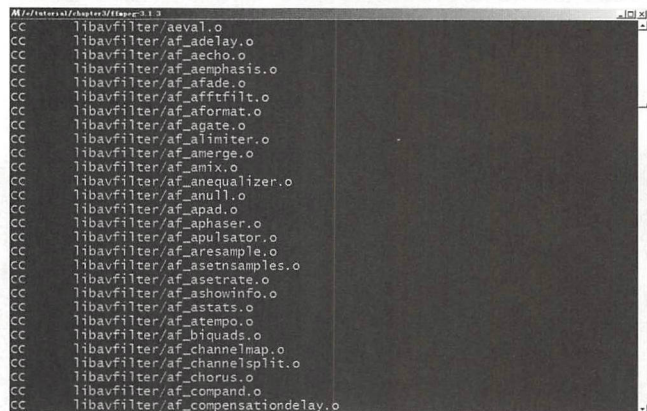


```

M:/tutorial/chapter3/ffmpeg-3.1.3
Administrator@QIWI-20170526 /e:/tutorial/chapter3/ffmpeg-3.1.3
$ ./build_android.sh
./configure: line 6204: [: =: unary operator expected
install prefix      E:/tutorial/chapter3/ffmpeg-3.1.3/android/arm
source path         .
C compiler          E:/android-ndk-r9d/toolchains/arm-linux-androideabi-4.
8/prebuilt/windows-x86_64/bin/arm-linux-androideabi-gcc
C library           bionic
host C compiler     gcc
host C library      gcc
ARCH               arm (armv5te)
big-endian         no
runtime cpu detection  yes
ARMv5TE enabled    yes
ARMv6 enabled      yes
ARMv6T2 enabled    yes
VFP enabled        yes
NEON enabled       yes
THUMB enabled      no
debug symbols      yes
strip symbols      yes
optimize for size  no
optimizations      yes
static             no
shared             yes
postprocessing support  yes
new filter support  yes
network support     yes

```

图 7-11 开始编译 FFmpeg



```

M:/tutorial/chapter3/ffmpeg-3.1.3
cc libavfilter/aeval.o
cc libavfilter/af_adelay.o
cc libavfilter/af_aecho.o
cc libavfilter/af_aemphasis.o
cc libavfilter/af_afade.o
cc libavfilter/af_afffilt.o
cc libavfilter/af_aformat.o
cc libavfilter/af_agate.o
cc libavfilter/af_alimiter.o
cc libavfilter/af_amerge.o
cc libavfilter/af_amix.o
cc libavfilter/af_anequalizer.o
cc libavfilter/af_anull.o
cc libavfilter/af_apad.o
cc libavfilter/af_aphaser.o
cc libavfilter/af_apulsator.o
cc libavfilter/af_aresample.o
cc libavfilter/af_asetnsamples.o
cc libavfilter/af_asetrate.o
cc libavfilter/af_ashowinfo.o
cc libavfilter/af_astats.o
cc libavfilter/af_atempo.o
cc libavfilter/af_biquads.o
cc libavfilter/af_channelmap.o
cc libavfilter/af_channelsoft.o
cc libavfilter/af_chorus.o
cc libavfilter/af_compand.o
cc libavfilter/af_compensationdelay.o

```

图 7-12 编译 FFmpeg 中间过程



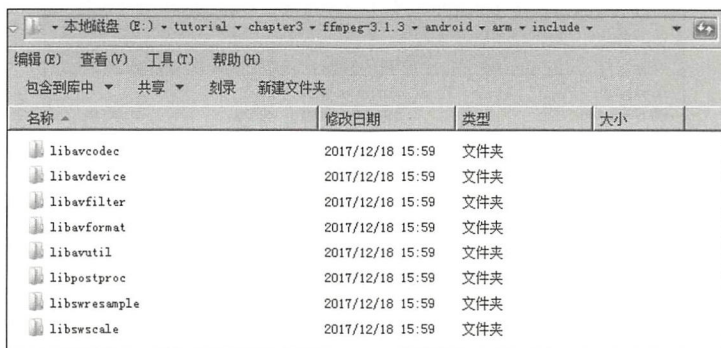
```

M:/tutorial/chapter3/ffmpeg-3.1.3
INSTALL libavutil/pixdesc.h
INSTALL libavutil/pixelutils.h
INSTALL libavutil/pixfmt.h
INSTALL libavutil/random_seed.h
INSTALL libavutil/rc4.h
INSTALL libavutil/rational.h
INSTALL libavutil/replaygain.h
INSTALL libavutil/ripemd.h
INSTALL libavutil/samplefmt.h
INSTALL libavutil/sha.h
INSTALL libavutil/sha512.h
INSTALL libavutil/stereo3d.h
INSTALL libavutil/threadmessage.h
INSTALL libavutil/time.h
INSTALL libavutil/timexcode.h
INSTALL libavutil/timestamp.h
INSTALL libavutil/tree.h
INSTALL libavutil/twofish.h
INSTALL libavutil/version.h
INSTALL libavutil/xtea.h
INSTALL libavutil/tea.h
INSTALL libavutil/120.h
INSTALL libavutil/avconfig.h
INSTALL libavutil/ffversion.h
INSTALL libavutil/libavutil.pc
Administrator@QIWI-20170526 /e:/tutorial/chapter3/ffmpeg-3.1.3
$

```

图 7-13 编译 FFmpeg 完成

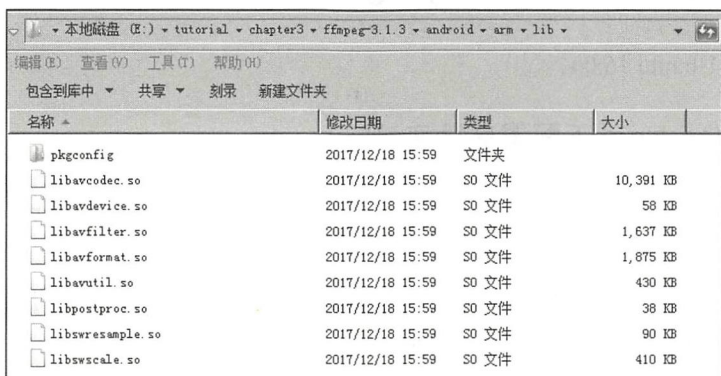
看到新生成的 android 目录下，多出一个 arm 文件夹，其中有一个头文件目录，如图 7-14 所示。



名称	修改日期	类型	大小
libavcodec	2017/12/18 15:59	文件夹	
libavdevice	2017/12/18 15:59	文件夹	
libavfilter	2017/12/18 15:59	文件夹	
libavformat	2017/12/18 15:59	文件夹	
libavutil	2017/12/18 15:59	文件夹	
libpostproc	2017/12/18 15:59	文件夹	
libswresample	2017/12/18 15:59	文件夹	
libswscale	2017/12/18 15:59	文件夹	

图 7-14 编译生成头文件目录

同时在 lib 目录下，生成了 8 个 .so 动态库，如图 7-15 所示，你也可以根据自身需要禁用一些配置。



名称	修改日期	类型	大小
pkgconfig	2017/12/18 15:59	文件夹	
libavcodec.so	2017/12/18 15:59	SO 文件	10,391 KB
libavdevice.so	2017/12/18 15:59	SO 文件	58 KB
libavfilter.so	2017/12/18 15:59	SO 文件	1,637 KB
libavformat.so	2017/12/18 15:59	SO 文件	1,875 KB
libavutil.so	2017/12/18 15:59	SO 文件	430 KB
libpostproc.so	2017/12/18 15:59	SO 文件	38 KB
libswresample.so	2017/12/18 15:59	SO 文件	90 KB
libswscale.so	2017/12/18 15:59	SO 文件	410 KB

图 7-15 编译生成动态库文件

7.2.6 编译静态库.a

如果想生成静态库.a，只需要改动一小块代码就行了，即把原来的如下配置项

```
--enable-shared \
--disable-static \
```

替换成

```
--disable-shared \
--enable-static \
```

然后重新执行 `./build_android.sh`，这样就可以生成静态库文件了，如图 7-16 所示。

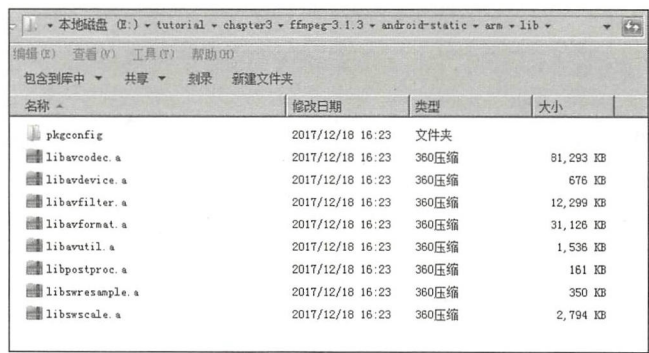


图 7-16 编译生成静态库文件

在 Windows 下编译 FFmpeg 动态库、静态库就介绍到这里。

7.3 在 Linux 下编译 FFmpeg

编译环境为 Ubuntu 16.04。

7.3.1 在 `/etc/profile.d` 下配置环境变量

在 Linux 的 `etc` 下有一个 `profile.d` 文件夹。很多人喜欢在 `/etc/profile` 下配置环境变量，笔者推荐在 `profile.d` 下配置，那么两者有什么区别呢？

第一，两个文件夹都用于配置环境变量，`/etc/profile` 用于配置永久性的环境变量，即全局变量，而 `/etc/profile.d` 的设置对所有用户生效。

第二，`/etc/profile.d` 比 `/etc/profile` 好维护，如果不想要某个变量，直接删除 `/etc/profile.d` 下对应的 shell 脚本即可，不用像 `/etc/profile` 那样改动文件。

接下来看看如何在 `profile.d` 文件夹下配置环境变量。

在 Linux 下对每个用户都要指定各自的环境变量，其中包括可执行的 `path` 路径，这些路径决定了每个用户在执行时使用的命令工具。

在一般的情况下，可以在每个用户的环境变量里设定各自的 `path` 变量值，然后执行 `export PATH` 使其生效，但如果用户比较多，安装的命令工具也会越来越多，且除了用户自己可以使用这些工具，`root` 用户或其他用户也可以使用，这时在每个用户环境变量里进行设置就比较复杂了。



所以可以用另一种方法，在/etc/profile.d下创建一个 java_path.sh 脚本，脚本内容如下：

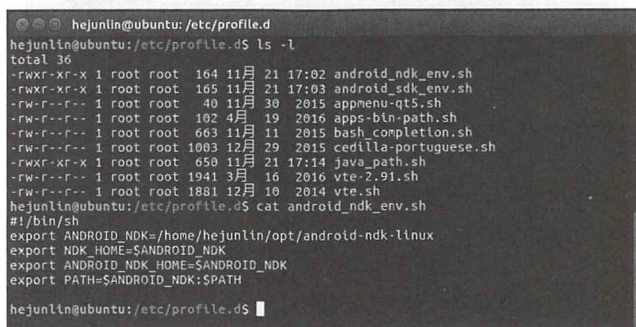
```
#set java environment
#JAVA_PATH=/opt/jdk1.7.0_65
JAVA_PATH=/home/hejunlin/opt/jdk1.8.0_121
#JAVA_PATH=/usr/lib/jvm/jdk
if [ -d "$JAVA_PATH/jre" -a -d "$JAVA_PATH/bin" ]; then
    export JAVA_HOME=$JAVA_PATH
    export ANDROID_JAVA_HOME=$JAVA_HOME
    export JRE_HOME=$JAVA_HOME/jre
    export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH

    if [ -z "$CLASSPATH" ]; then
        #export CLASSPATH=$JAVA_HOME/lib:$JRE_HOME/lib
        export CLASSPATH=.:$JAVA_HOME/lib:$JAVA_HOME/lib/tools.jar:$JAVA_HOME/
lib/dt.jar
    else
        #export CLASSPATH=$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
        export CLASSPATH=.:$JAVA_HOME/lib:$JAVA_HOME/lib/tools.jar:$JAVA_HOME/
lib/dt.jar:$CLASSPATH
    fi
fi
```

创建好后，可以通过 cat 命令进行查看：

```
# cat /etc/profile.d/java_path.sh
```

编译 FFmpeg，这时需要 NDK，所以最终配置 NDK 的 shell 脚本，如图 7-17 所示。



```
hejunlin@ubuntu: /etc/profile.d
hejunlin@ubuntu: /etc/profile.d$ ls -l
total 36
-rwxr-xr-x 1 root root 164 11月 21 17:02 android_ndk_env.sh
-rwxr-xr-x 1 root root 165 11月 21 17:03 android_sdk_env.sh
-rw-r--r-- 1 root root 40 11月 30 2015 appmenu-qt5.sh
-rw-r--r-- 1 root root 102 4月 19 2016 apps-bin-path.sh
-rw-r--r-- 1 root root 663 11月 11 2015 bash_completion.sh
-rw-r--r-- 1 root root 1003 12月 29 2015 cedilla-portuguese.sh
-rwxr-xr-x 1 root root 650 11月 21 17:14 java_path.sh
-rw-r--r-- 1 root root 1941 3月 16 2016 vte-2.91.sh
-rw-r--r-- 1 root root 1881 12月 10 2014 vte.sh
hejunlin@ubuntu: /etc/profile.d$ cat android_ndk_env.sh
#!/bin/sh
export ANDROID_NDK=/home/hejunlin/opt/android-ndk-linux
export NDK_HOME=$ANDROID_NDK
export ANDROID_NDK_HOME=$ANDROID_NDK
export PATH=$ANDROID_NDK:$PATH
hejunlin@ubuntu: /etc/profile.d$
```

图 7-17 配置 NDK 的 shell 脚本

可以将需要各用户执行的命令路径都写在该命令中，这样在每次操作系统启动后，会自动执行 java_path.sh 脚本，使所有的环境变量生效，让各用户都可以直接执行各自的命令。

以后再安装新软件，只需要将软件相关路径加入/etc/profile.d/java_path.sh 脚本，使脚本生



效后，所有用户都可以使用，不需要在多个地方重复添加。

7.3.2 开始编译 FFmpeg-3.1.3

编译前，在源码中修改 FFmpeg 的 configure 文件。由于编译出来的动态库文件名的版本号在.so 之后（例如“libavcodec.so.5.100.1”），而 Android 平台不能识别这样的文件名，所以需要修改该文件名。在 configure 文件中找到下面几行代码（在 3209~3212 行）：

```
SLIBNAME_WITH_MAJOR='$(SLIBNAME).$(LIBMAJOR)'
LIB_INSTALL_EXTRA_CMD='$(RANLIB)"$(LIBDIR)/$(LIBNAME) "'
SLIB_INSTALL_NAME='$(SLIBNAME_WITH_VERSION)'
SLIB_INSTALL_LINKS='$(SLIBNAME_WITH_MAJOR)$ (SLIBNAME)'
```

替换为下面的内容

```
SLIBNAME_WITH_MAJOR='$(SLIBPREF)$(FULLNAME)-$(LIBMAJOR)$(SLIBSUF)'
LIB_INSTALL_EXTRA_CMD='$(RANLIB)"$(LIBDIR)/$(LIBNAME) "'
SLIB_INSTALL_NAME='$(SLIBNAME_WITH_MAJOR)'
SLIB_INSTALL_LINKS='$(SLIBNAME)'
```

修改后的 configure 文件内容如图 7-18 所示。

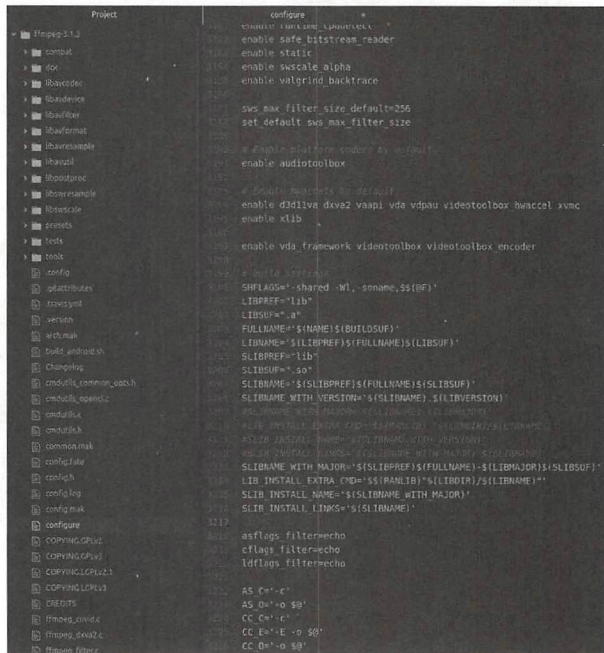


图 7-18 修改后的 configure 文件内容



7.3.3 编写 shell 脚本

编译 shell 脚本内容如图 7-19 所示。

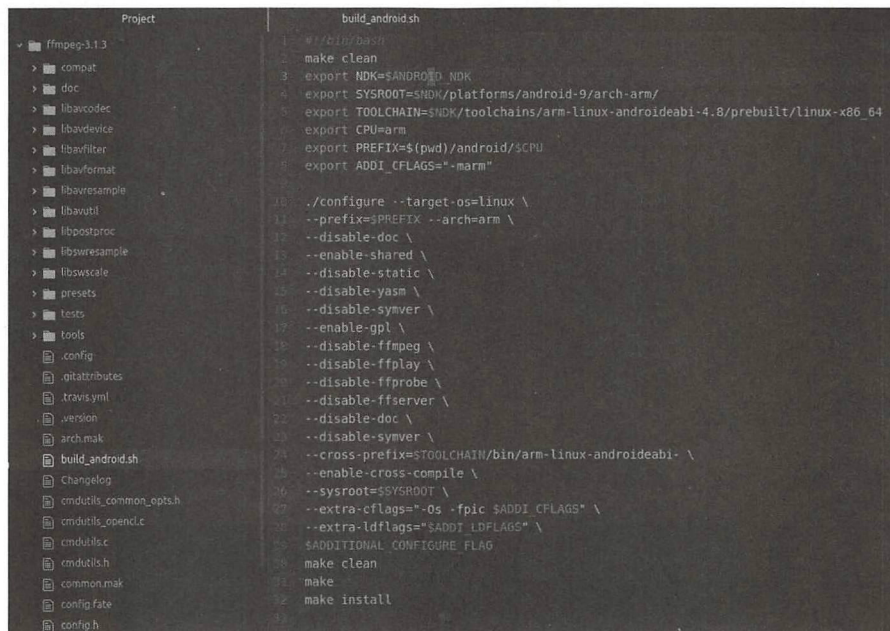


图 7-19 编译 shell 脚本内容

build_android.sh 文件完整代码如下：

```
#!/bin/bash
make clean
export NDK=$ANDROID_NDK
export SYSROOT=$NDK/platforms/android-9/arch-arm/
export TOOLCHAIN=$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/
linux-x86_64
export CPU=arm
export PREFIX=$(pwd)/android-static/$CPU
export ADDI_CFLAGS="-marm"
./configure --target-os=linux \
--prefix=$PREFIX --arch=arm \
--disable-doc \
--disable-shared \
--enable-static \
--disable-yasm \
--disable-symver \
--enable-gpl \
```




```
--disable-ffmpeg \  
--disable-ffplay \  
--disable-ffprobe \  
--disable-ffserver \  
--disable-doc \  
--disable-symver \  
--cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \  
--enable-cross-compile \  
--sysroot=$SYSROOT \  
--extra-cflags="-Os -fpic $ADDI_CFLAGS" \  
--extra-ldflags="$ADDI_LDFLAGS" \  
$ADDITIONAL_CONFIGURE_FLAG  
make clean  
make  
make install
```

7.3.4 编译动态库.so

开始执行./build_android.sh，出现如图 7-20 所示的界面，表示开始编译 FFmpeg。



```
hejunlin@ubuntu: ~/tutorial/chapter-3/ffmpeg-3.1.3  
cc libavfilter/af_replaygain.o  
cc libavfilter/af_sidechaincompress.o  
cc libavfilter/af_silencedetect.o  
cc libavfilter/af_silenceremove.o  
cc libavfilter/af_stereotools.o  
cc libavfilter/af_stereowiden.o  
cc libavfilter/af_tremolo.o  
cc libavfilter/af_vibrato.o  
cc libavfilter/af_volume.o  
cc libavfilter/af_volumedetect.o  
cc libavfilter/allfilters.o  
cc libavfilter/asink_anullsink.o  
cc libavfilter/asrc_anois-src.o  
cc libavfilter/asrc_anullsrc.o  
cc libavfilter/asrc_sine.o  
cc libavfilter/audio.o  
cc libavfilter/avf_ahistogram.o  
cc libavfilter/avf_aphasemeter.o  
cc libavfilter/avf_avectroscope.o  
cc libavfilter/avf_concat.o  
cc libavfilter/avf_showcqt.o  
cc libavfilter/avf_showfreqs.o  
cc libavfilter/avf_showspectrum.o  
cc libavfilter/avf_showvolume.o  
cc libavfilter/avf_showwaves.o  
cc libavfilter/avfilter.o  
cc libavfilter/avfiltergraph.o  
cc libavfilter/bbox.o  
cc libavfilter/buffersink.o  
cc libavfilter/buffersrc.o  
cc libavfilter/colormappedsp.o
```

图 7-20 开始编译 FFmpeg

编译 FFmpeg 中间过程，如图 7-21 所示。

编译 FFmpeg 完成，如图 7-22 所示。

这时发现 ffmpeg 目录下多了一个 android 文件夹（如图 7-23 所示），下面的 arm 子文件夹



下包含 include 头文件。

```
hejunlin@ubuntu: ~/tutorial/chapter-3/ffmpeg-3.1.3
CC      libavformat/tty.o
CC      libavformat/txd.o
CC      libavformat/udp.o
CC      libavformat/uncodedframecrcenc.o
CC      libavformat/unix.o
CC      libavformat/url.o
CC      libavformat/urldecode.o
CC      libavformat/utls.o
CC      libavformat/v210.o
CC      libavformat/vag.o
CC      libavformat/vc1dec.o
CC      libavformat/vc1test.o
CC      libavformat/vc1testenc.o
CC      libavformat/vivo.o
CC      libavformat/voc.o
CC      libavformat/voc_packet.o
CC      libavformat/vocdec.o
CC      libavformat/vocenc.o
CC      libavformat/vorbiscomment.o
CC      libavformat/vpcc.o
CC      libavformat/vpk.o
CC      libavformat/vplayerdec.o
CC      libavformat/vqf.o
CC      libavformat/w64.o
CC      libavformat/wavdec.o
CC      libavformat/wavenc.o
CC      libavformat/wc3movie.o
CC      libavformat/webm_chunk.o
CC      libavformat/webmdashenc.o
CC      libavformat/webpenc.o
CC      libavformat/webvttdec.o
```

图 7-21 编译 FFmpeg 中间过程

```
hejunlin@ubuntu: ~/tutorial/chapter-3/ffmpeg-3.1.3
INSTALL libavutil/nd5.h
INSTALL libavutil/nem.h
INSTALL libavutil/notion_vector.h
INSTALL libavutil/nurmur3.h
INSTALL libavutil/opt.h
INSTALL libavutil/parseutils.h
INSTALL libavutil/pixdesc.h
INSTALL libavutil/pixelutils.h
INSTALL libavutil/pixfmt.h
INSTALL libavutil/random_seed.h
INSTALL libavutil/rc4.h
INSTALL libavutil/rational.h
INSTALL libavutil/replaygain.h
INSTALL libavutil/ripend.h
INSTALL libavutil/samplefmt.h
INSTALL libavutil/sha.h
INSTALL libavutil/sha512.h
INSTALL libavutil/stereo3d.h
INSTALL libavutil/threadmessage.h
INSTALL libavutil/time.h
INSTALL libavutil/timcode.h
INSTALL libavutil/timestamp.h
INSTALL libavutil/tree.h
INSTALL libavutil/twofish.h
INSTALL libavutil/version.h
INSTALL libavutil/xtea.h
INSTALL libavutil/tea.h
INSTALL libavutil/lzo.h
INSTALL libavutil/avconfig.h
INSTALL libavutil/ffversion.h
INSTALL libavutil/libavutil.pc
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3$
```

图 7-22 编译 FFmpeg 完成

通过 `ls -l` 命令看看对应的文件，可以发现 8 个与 FFmpeg 相关的 .so 文件生成了，如图 7-24 所示。



```

hejunlin@ubuntu: ~/tutorial/chapter-3/ffmpeg-3.1.3
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3$ ls -l
total 2220
drwxrwxr-x 3 hejunlin hejunlin 4096 12月 18 14:01 android
-rwxr-xr-x 1 hejunlin hejunlin 827 6月 27 2016 arch.mak
-rwxrwxrwx 1 hejunlin hejunlin 768 12月 18 12:43 build_android.sh
-rwxr-xr-x 1 hejunlin hejunlin 56628 8月 26 2016 Changelog
-rwxr-xr-x 1 hejunlin hejunlin 72929 8月 26 2016 cmdutils.c
-rwxr-xr-x 1 hejunlin hejunlin 2898 3月 29 2016 cmdutils_common_opts.h
-rwxr-xr-x 1 hejunlin hejunlin 18987 8月 9 2016 cmdutils.h
-rwxr-xr-x 1 hejunlin hejunlin 10439 6月 27 2016 cmdutils_opencl.c
-rwxr-xr-x 1 hejunlin hejunlin 4504 6月 27 2016 common.mak
drwxr-xr-x 11 hejunlin hejunlin 4096 12月 18 12:56 compat
-rw-rw-r-- 1 hejunlin hejunlin 626 12月 18 13:54 config.fate
-rw-rw-r-- 1 hejunlin hejunlin 68039 12月 18 12:55 config.h
-rw-rw-r-- 1 hejunlin hejunlin 445659 12月 18 13:54 config.log
-rw-rw-r-- 1 hejunlin hejunlin 61461 12月 18 13:54 config.mak
-rwxr-xr-x 1 hejunlin hejunlin 222625 12月 18 13:25 configure
-rwxr-xr-x 1 hejunlin hejunlin 18092 3月 29 2016 COPYING.GPLV2
-rwxr-xr-x 1 hejunlin hejunlin 35147 3月 29 2016 COPYING.GPLV3
-rwxr-xr-x 1 hejunlin hejunlin 26526 3月 29 2016 COPYING.LGPLv2.1
-rwxr-xr-x 1 hejunlin hejunlin 7651 3月 29 2016 COPYING.LGPLv3
-rwxr-xr-x 1 hejunlin hejunlin 274 3月 29 2016 CREDITS
drwxr-xr-x 4 hejunlin hejunlin 4096 12月 18 12:55 doc
-rwxr-xr-x 1 hejunlin hejunlin 160065 8月 9 2016 ffmpeg.c
-rwxr-xr-x 1 hejunlin hejunlin 7213 6月 27 2016 ffmpeg_cuid.c
-rwxr-xr-x 1 hejunlin hejunlin 15057 6月 27 2016 ffmpeg_dxva2.c
-rwxr-xr-x 1 hejunlin hejunlin 42202 6月 27 2016 ffmpeg_filter.c
-rwxr-xr-x 1 hejunlin hejunlin 17915 6月 27 2016 ffmpeg.h
-rwxr-xr-x 1 hejunlin hejunlin 139573 6月 27 2016 ffmpeg_opt.c
-rwxr-xr-x 1 hejunlin hejunlin 7947 6月 27 2016 ffmpeg_qsv.c
-rwxr-xr-x 1 hejunlin hejunlin 17999 6月 27 2016 ffmpeg_vaapi.c
-rwxr-xr-x 1 hejunlin hejunlin 4469 6月 27 2016 ffmpeg_vdpau.c

```

图 7-23 编译生成的 android 文件夹

```

hejunlin@ubuntu: ~/tutorial/chapter-3/ffmpeg-3.1.3/android/arm/lib
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android$ cd arm
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android/arm$ ls -l
total 8
drwxrwxr-x 10 hejunlin hejunlin 4096 12月 18 14:01 include
drwxrwxr-x 3 hejunlin hejunlin 4096 12月 18 14:01 lib
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android/arm$ cd lib/
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android/arm/lib$ ls
libavcodec-57.so  libavfilter-6.so  libpostproc-54.so  libswscale-4.so
libavcodec.so     libavformat-57.so  libpostproc.so     pkgconfig
libavdevice-57.so libavutil-55.so    libswresample-2.so
libavfilter-6.so  libavutil.so       libswscale-4.so
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android/arm/lib$ ls -l
total 14952
-rwxr-xr-x 1 hejunlin hejunlin 10639408 12月 18 14:01 libavcodec-57.so
lrwxrwxrwx 1 hejunlin hejunlin 16 12月 18 14:01 libavcodec.so -> libavcodec-57.so
-rwxr-xr-x 1 hejunlin hejunlin 63408 12月 18 14:01 libavdevice-57.so
lrwxrwxrwx 1 hejunlin hejunlin 17 12月 18 14:01 libavdevice.so -> libavdevice-57.so
-rwxr-xr-x 1 hejunlin hejunlin 1675820 12月 18 14:01 libavfilter-6.so
lrwxrwxrwx 1 hejunlin hejunlin 16 12月 18 14:01 libavfilter.so -> libavfilter-6.so
-rwxr-xr-x 1 hejunlin hejunlin 1919008 12月 18 14:01 libavformat-57.so
lrwxrwxrwx 1 hejunlin hejunlin 17 12月 18 14:01 libavformat.so -> libavformat-57.so
-rwxr-xr-x 1 hejunlin hejunlin 439720 12月 18 14:01 libavutil-55.so
lrwxrwxrwx 1 hejunlin hejunlin 15 12月 18 14:01 libavutil.so -> libavutil-55.so
-rwxr-xr-x 1 hejunlin hejunlin 38192 12月 18 14:01 libpostproc-54.so
lrwxrwxrwx 1 hejunlin hejunlin 17 12月 18 14:01 libpostproc.so -> libpostproc-54.so
-rwxr-xr-x 1 hejunlin hejunlin 91540 12月 18 14:01 libswresample-2.so
lrwxrwxrwx 1 hejunlin hejunlin 18 12月 18 14:01 libswresample.so -> libswresample-2.so
-rwxr-xr-x 1 hejunlin hejunlin 419220 12月 18 14:01 libswscale-4.so
lrwxrwxrwx 1 hejunlin hejunlin 15 12月 18 14:01 libswscale.so -> libswscale-4.so
drwxrwxr-x 2 hejunlin hejunlin 4096 12月 18 14:01 pkgconfig
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android/arm/lib$

```

图 7-24 编译生成动态库文件

7.3.5 编译静态库.a

同前面一样，如果要编译静态库.a，把原来的如下配置项

```
--enable-shared \
--disable-static \
```

替换成

```
--disable-shared \
--enable-static \
```




然后重新执行./build_android.sh，就可以生成 8 个与 FFmpeg 相关的静态库文件，如图 7-25 所示。

```

hejunlin@ubuntu: ~/tutorial/chapter-3/ffmpeg-3.1.3/android-static/arm/lib
drwxr-xr-x 7 hejunlin hejunlin 4096 12月 18 14:22 libswscale
-rwxr-xr-x 1 hejunlin hejunlin 4502 6月 27 2016 LICENSE.md
-rwxr-xr-x 1 hejunlin hejunlin 27085 6月 27 2016 MAINTAINERS
-rwxr-xr-x 1 hejunlin hejunlin 6562 6月 27 2016 Makefile
drwxr-xr-x 2 hejunlin hejunlin 4096 3月 29 2016 presets
-rwxr-xr-x 1 hejunlin hejunlin 1941 3月 27 2016 README.md
-rwxr-xr-x 1 hejunlin hejunlin 6 8月 26 2016 RELEASE
-rwxr-xr-x 1 hejunlin hejunlin 847 6月 27 2016 RELEASE_NOTES
drwxr-xr-x 7 hejunlin hejunlin 4096 6月 27 2016 tests
drwxr-xr-x 2 hejunlin hejunlin 4096 6月 27 2016 tools
-rwxr-xr-x 1 hejunlin hejunlin 6 8月 26 2016 VERSION
-rwxr-xr-x 1 hejunlin hejunlin 1933 6月 27 2016 version.sh
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3$ cd android-static/
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android-static$ cd arm/lib/
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android-static/arm/lib$ ls -l
total 130412
-rw-r--r-- 1 hejunlin hejunlin 83285330 12月 18 14:22 libavcodec.a
-rw-r--r-- 1 hejunlin hejunlin 764036 12月 18 14:22 libavdevice.a
-rw-r--r-- 1 hejunlin hejunlin 12607768 12月 18 14:22 libavfilter.a
-rw-r--r-- 1 hejunlin hejunlin 31908102 12月 18 14:22 libavformat.a
-rw-r--r-- 1 hejunlin hejunlin 1576510 12月 18 14:22 libavutil.a
-rw-r--r-- 1 hejunlin hejunlin 164340 12月 18 14:22 libpostproc.a
-rw-r--r-- 1 hejunlin hejunlin 358648 12月 18 14:22 libswresample.a
-rw-r--r-- 1 hejunlin hejunlin 2861518 12月 18 14:22 libswscale.a
drwxrwxr-x 2 hejunlin hejunlin 4096 12月 18 14:22 pkgconfig
hejunlin@ubuntu:~/tutorial/chapter-3/ffmpeg-3.1.3/android-static/arm/lib$

```

图 7-25 编译生成静态库文件

到这里，我们就完成了在 Linux 下编译 FFmpeg 动态库、静态库。

7.4 在 Mac OS 下编译 FFmpeg

前面分别在 Windows 下和 Linux 下编译了 FFmpeg，并且生成了动态库和静态库文件，而本节将在 Mac OS 下编译 FFmpeg。

7.4.1 下载源码及配置环境变量

下载 FFmpeg-3.1.3，如图 7-26 所示。

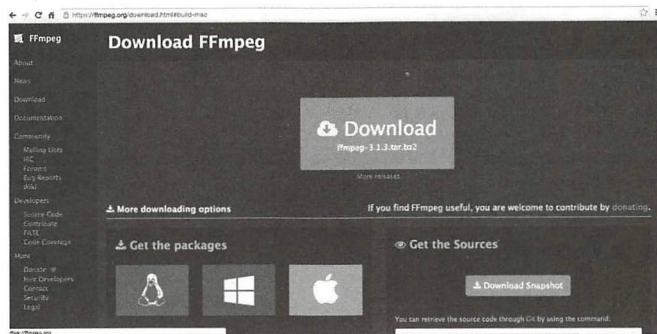


图 7-26 FFmpeg 官网对应的 3.1.3 版本下载页面



双击下载的安装包进行解压缩，得到一个文件夹，如图 7-27 所示。

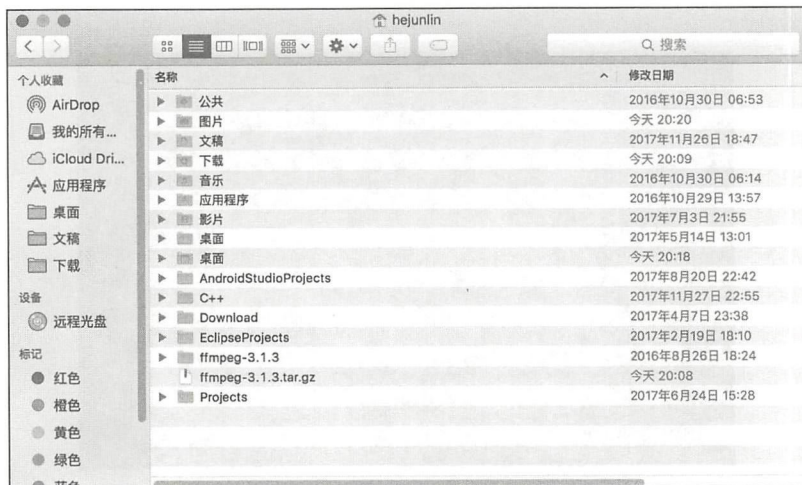


图 7-27 解压缩 FFmpeg-3.1.3 安装包

然后同样下载 NDK 安装包，安装包解压缩后，同样得到一个文件夹，如图 7-28 所示。

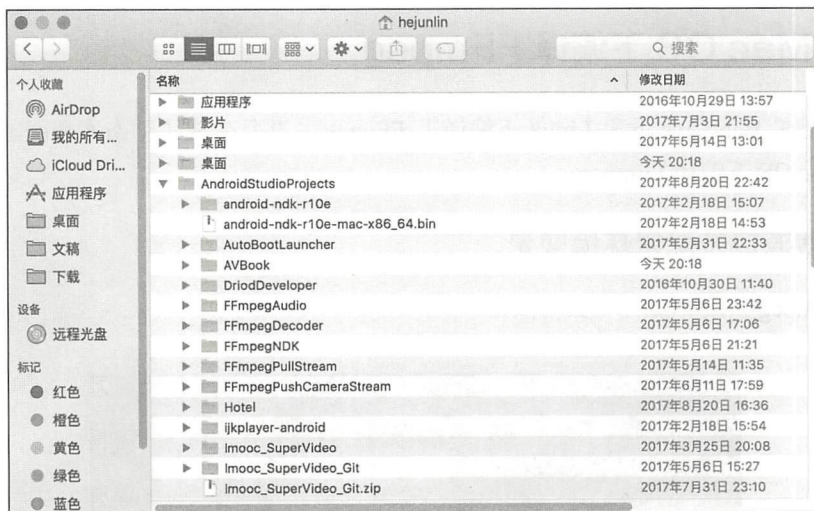


图 7-28 解压缩 NDK 安装包

接下来配置 NDK 环境，如图 7-29 所示。

- 启动终端 Terminal。
- 进入当前用户的 home 目录（输入 `cd ~` 或 `/Users/YourUserName`）。



- 创建.bash_profile 文件（输入 touch .bash_profile）。
- 编辑.bash_profile 文件（输入 open -e .bash_profile）。



图 7-29 配置 NDK 环境

为了配置 NDK 开发环境，需要输入 Mac OS 下的 NDK 的目录，同样配置 Android SDK 环境也使用这种方式，最终的.bash_profile 文件内容如图 7-30 所示。



图 7-30 最终的.bash_profile 文件内容

下面是刚刚配置的内容，读者在自己的机器上配置并更改相应的路径就可以：

```
export PATH=${PATH}:/Users/hejunlin/Library/Android/sdk/tools
export PATH=${PATH}:/Users/hejunlin/Library/Android/sdk/platform-tools
```



```
export NDK_ROOT=/Users/hejunlin/AndroidStudioProjects/android-ndk-r10e
export PATH=$PATH:$NDK_ROOT
```

接着执行后面的步骤，使配置文件生效。

- 保存文件，关闭.bash_profile。
- 更新刚配置的环境变量，输入 source .bash_profile。
- 看看刚刚设置的环境变量。

离开了编辑器后，在终端输入\$PATH 并且按下 Enter 键来确认是否编辑成功，此时应该会出现所有环境变量（以:号分隔）。最终配置文件内容如图 7-31 所示。

```
hejunlindeMacBook-Pro:ffmpeg-3.1.3 hejunlin$ $PATH
-bash: /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Frameworks/Mono.framework/Versions/Current/Commands:/Users/hejunlin/Documents/local/apache-tomcat-7.0.72/bin:/Users/hejunlin/Library/Android/sdk/tools:/Users/hejunlin/Library/Android/sdk/platform-tools:/Users/hejunlin/AndroidStudioProjects/android-ndk-r10e: No such file or directory
hejunlindeMacBook-Pro:ffmpeg-3.1.3 hejunlin$
```

图 7-31 最终配置文件内容

这时表明配置成功。接下来测试 NDK 是否能正常编译 JNI。

- (1) 进入终端 NDK 下面的 samples 目录。
- (2) 输入 cd hello-jni/命令，按下 Enter 键，然后执行 ndk-build 命令。

出现如图 7-32 所示的界面代表配置成功。

```
hejunlindeMacBook-Pro:~ hejunlin$ cd /Users/hejunlin/AndroidStudioProjects/android-ndk-r10e
hejunlindeMacBook-Pro:android-ndk-r10e hejunlin$ cd samples/
hejunlindeMacBook-Pro:samples hejunlin$ cd hello-jni/
hejunlindeMacBook-Pro:hello-jni hejunlin$ ndk-build
[arm64-v8a] Gdbserver : [aarch64-linux-android-4.9] libs/arm64-v8a/gdbserver
[arm64-v8a] Gdbsetup : libs/arm64-v8a/gdb.setup
[x86_64] Gdbserver : [x86_64-linux-android-4.9] libs/x86_64/gdbserver
[x86_64] Gdbsetup : libs/x86_64/gdb.setup
[mips64] Gdbserver : [mips64el-linux-android-4.9] libs/mips64/gdbserver
[mips64] Gdbsetup : libs/mips64/gdb.setup
[armeabi-v7a] Gdbserver : [arm-linux-androideabi-4.8] libs/armeabi-v7a/gdbserver
[armeabi-v7a] Gdbsetup : libs/armeabi-v7a/gdb.setup
[armeabi] Gdbserver : [arm-linux-androideabi-4.8] libs/armeabi/gdbserver
[armeabi] Gdbsetup : libs/armeabi/gdb.setup
[x86] Gdbserver : [x86-linux-android-4.8] libs/x86/gdbserver
[x86] Gdbsetup : libs/x86/gdb.setup
[mips] Gdbserver : [mipsel-linux-android-4.8] libs/mips/gdbserver
[mips] Gdbsetup : libs/mips/gdb.setup
[arm64-v8a] Install : libhello-jni.so => libs/arm64-v8a/libhello-jni.so
[x86_64] Install : libhello-jni.so => libs/x86_64/libhello-jni.so
[mips64] Install : libhello-jni.so => libs/mips64/libhello-jni.so
[armeabi-v7a] Install : libhello-jni.so => libs/armeabi-v7a/libhello-jni.so
[armeabi] Install : libhello-jni.so => libs/armeabi/libhello-jni.so
[x86] Install : libhello-jni.so => libs/x86/libhello-jni.so
[mips] Install : libhello-jni.so => libs/mips/libhello-jni.so
hejunlindeMacBook-Pro:hello-jni hejunlin$
```

图 7-32 配置成功

7.4.2 开始编译 FFmpeg-3.1.3

编译前，在源码中修改 FFmpeg 的 configure 文件。由于编译出来的动态库文件名的版本号在.so 之后（例如“libavcodec.so.5.100.1”），而 Android 平台不能识别这样的文件名，所以需要修改该文件名。在 configure 文件中找到下面几行代码（在 3209~3212 行）：

```
SLIBNAME_WITH_MAJOR='${SLIBNAME}.${LIBMAJOR}'
LIB_INSTALL_EXTRA_CMD='${RANLIB}' "${LIBDIR}/${LIBNAME}"
SLIB_INSTALL_NAME='${SLIBNAME_WITH_VERSION}'
SLIB_INSTALL_LINKS='${SLIBNAME_WITH_MAJOR}${SLIBNAME}'
```

替换为下面的内容：

```
SLIBNAME_WITH_MAJOR='${SLIBPREF}${FULLNAME}-${LIBMAJOR}${SLIBSUF}'
LIB_INSTALL_EXTRA_CMD='${RANLIB}' "${LIBDIR}/${LIBNAME}"
SLIB_INSTALL_NAME='${SLIBNAME_WITH_MAJOR}'
SLIB_INSTALL_LINKS='${SLIBNAME}'
```

修改后的 configure 文件内容如图 7-33 所示。

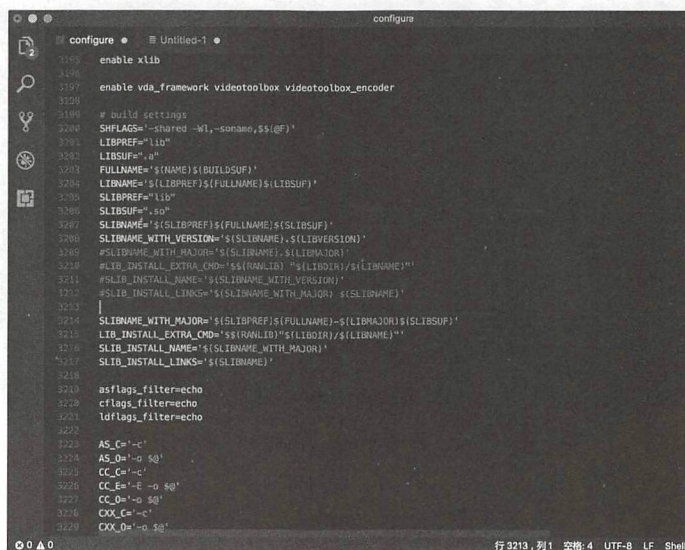


图 7-33 修改后的 configure 文件内容

7.4.3 编写 shell 脚本

接下来开始编写 shell 脚本。

这里有一个“坑”需要注意，笔者在 Windows 下用记事本程序写了一个 sh 脚本，但是一直报如图 7-34 所示的编译错误。

```

bogon:~ hejunlin$ cd /Users/hejunlin/Downloads/ffmpeg-3.1.3
bogon:ffmpeg-3.1.3 hejunlin$ sudo chmod +x build_android.sh
Password:
bogon:ffmpeg-3.1.3 hejunlin$ ./build_android.sh
-bash: ./build_android.sh: /bin/bash^M: bad interpreter: No such file or directo
ry

```

图 7-34 编译报错

这种情况的发生应该有如下的两个原因。

- 在 Windows 系统中用文本编辑工具修改过参数变量，在保存的时候没注意编码格式。
- 也有可能在 Vim 里进行修改时，第 1 行末尾按下了 Ctrl+V 快捷键。

为了避开这个“坑”，笔者找到 ffmpeg 下名为 version.sh 的 shell 脚本，复制了一份，并重命名为 build_android.sh，如图 7-35 所示。

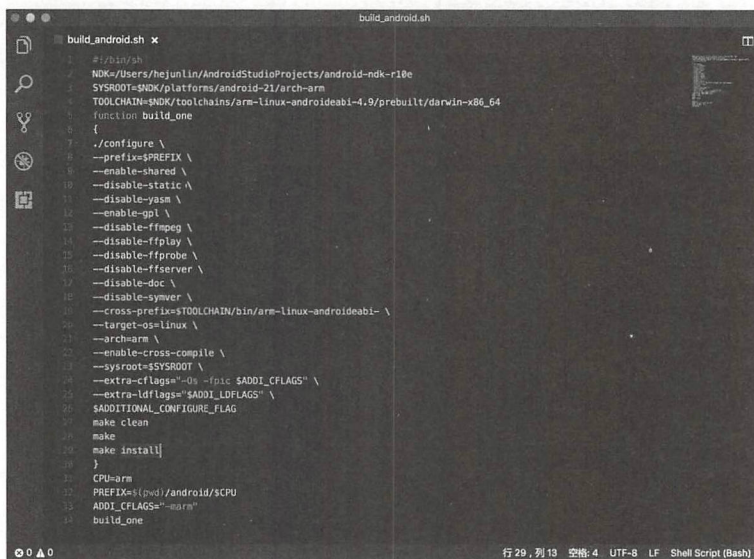


图 7-35 新的 build_Android.sh 文件

脚本内容如下：

```

#!/bin/sh
NDK=/Users/hejunlin/Downloads/android-ndk-r10e
SYSROOT=$NDK/platforms/android-21/arch-arm
TOOLCHAIN=$NDK/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-
x86_64
function build_one
{
./configure \
--prefix=$PREFIX \

```



```

--enable-shared \
--disable-static \
--disable-doc \
--disable-ffmpeg \
--disable-ffplay \
--disable-ffprobe \
--disable-ffserver \
--disable-avdevice \
--disable-doc \
--disable-symver \
--cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \
--target-os=linux \
--arch=arm \
--enable-cross-compile \
--sysroot=$SYSROOT \
--extra-cflags="-Os -fpic $ADDI_CFLAGS" \
--extra-ldflags="$ADDI_LDFLAGS" \
$ADDITIONAL_CONFIGURE_FLAG
make clean
make
make install
}
CPU=arm
PREFIX=$(pwd) /android/$CPU
ADDI_CFLAGS="-marm"
build_one

```

如果读者要编译该脚本，记得修改前 3 行代码，使其对应到自己机器上的环境。

接着在终端输入 `./build_android.sh`，开始运行这个 shell 脚本，如图 7-36 所示。

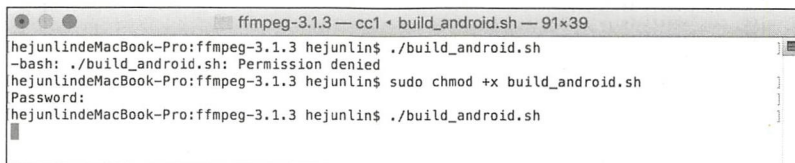


图 7-36 正常运行脚本

7.4.4 编译动态库.so

开始进行自动编译：

如果你的脚本有问题，有时候甚至只是多了一个空格，也会出现如图 7-37 所示的界面。

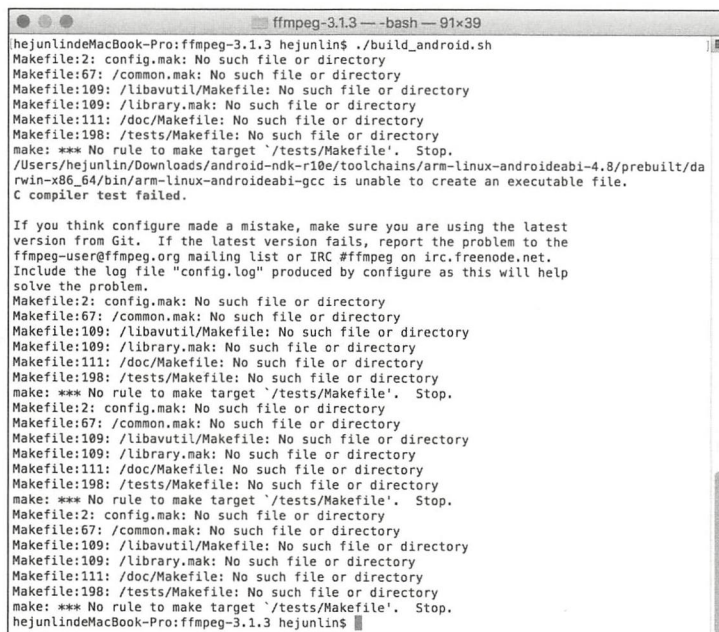


图 7-37 运行脚本异常

修改后正常运行脚本，会出现如图 7-38 所示的界面。

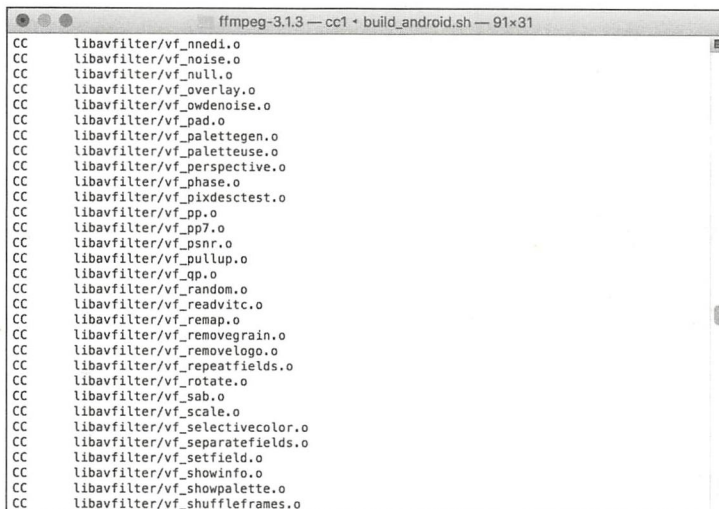


图 7-38 修改后正常运行脚本

运行 FFmpeg 中间过程，如图 7-39 所示。

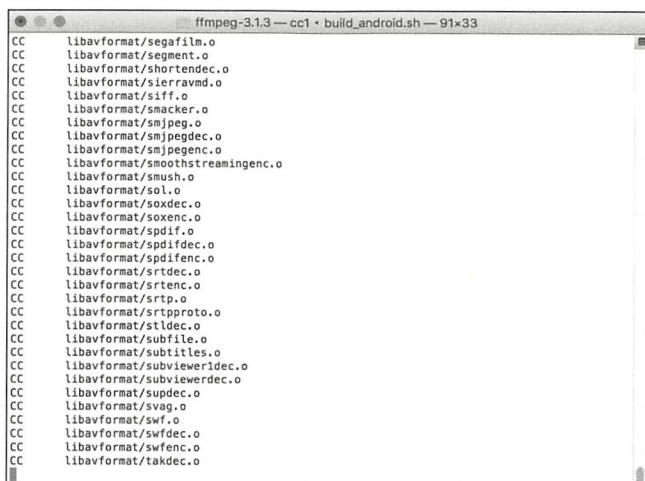


图 7-39 运行 FFmpeg 中间过程

喝杯咖啡的时间就编译好了，其结果如图 7-40 所示。

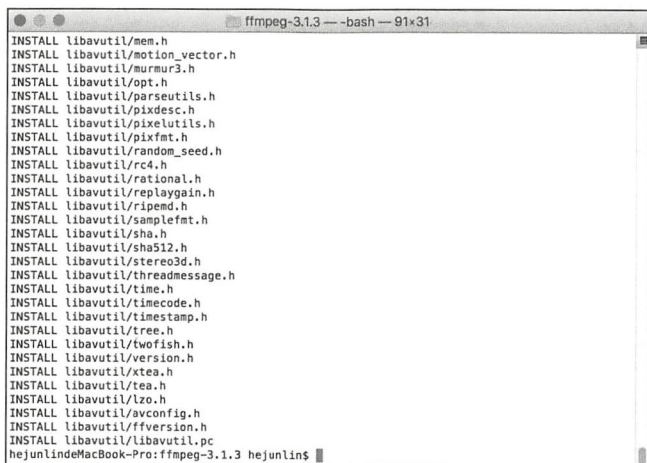


图 7-40 编译 FFmpeg 完成

这时会发现 ffmpeg 下多了一个 android 文件夹，展开可以发现对应的头文件和 8 个与 FFmpeg 相关的.so 动态库都已经生成，如图 7-41 所示。

7.4.5 编译静态库.a

同前面一样，如果要编译静态库.a，把原来的如下配置项

```
--enable-shared \
```



```
--disable-static \
```

替换成

```
--disable-shared \  
--enable-static \
```

然后重新执行./build_android.sh，就可以生成静态库了，如图 7-42 所示。

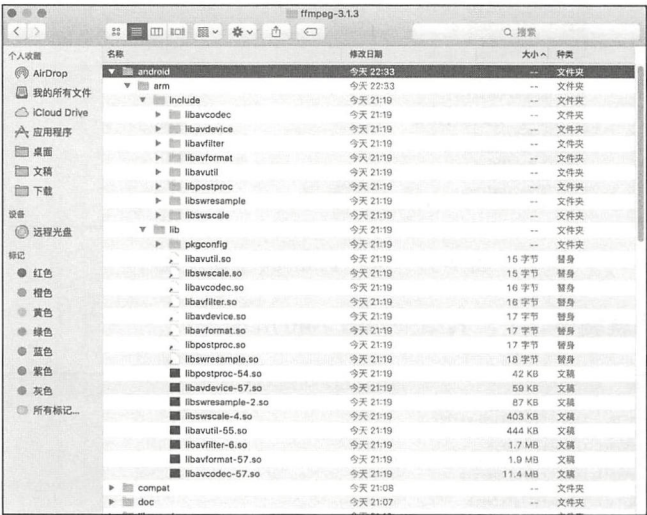


图 7-41 编译生成动态库文件

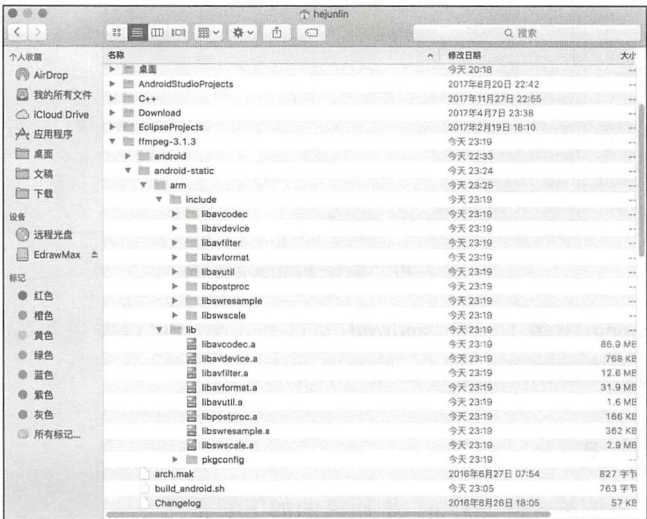


图 7-42 编译生成静态库文件

到这里，我们就完成了在 Mac OS 下编译 FFmpeg 动态库、静态库。

7.5 FFmpeg 常用命令

7.5.1 改变帧率、码率和文件大小

改变帧率，我们可以使用如下命令：

```
ffmpeg -i input.avi -r 30 output.mp4
```

码率（也叫作比特率或数据速率）是一个决定整体音频或视频质量的参数。它指定每个时间单位处理的比特数，在 FFmpeg 中，码率以每秒比特数表示。

码率决定使用多少位存储 1s 内编码的流数据，它是用 -b 选项设置区分音频流和视频流的，推荐使用 -b:a 或 -b:v。例如，设置一个 overall，每秒 1.5 兆比特（Mb/s），我们可以使用下面的命令：

```
ffmpeg -i film.avi -b 1.5M film.mp4
```

为了将输出文件的大小保持在一定的值以下，可以使用 -fs 选项（文件大小的缩写），单位是 byte。例如，要指定最大输出文件大小为 10MB，可以使用如下命令：

```
ffmpeg -i input.avi -fs 10MB output.mp4
```

下面介绍文件大小的计算方式。

编码输出的最终文件大小是音频流和视频流文件大小的总和。视频流文件大小的计算方式如下：

$$\text{video_size} = \text{video_bitrate} \times \text{time_in_seconds} / 8 \quad (\text{以 byte 为单位})$$

如果音频没有压缩，音频流文件大小的计算方式如下：

$$\text{audio_size} = \text{sampling_rate} \times \text{bit_depth} \times \text{channels} \times \text{time_in_seconds} / 8$$

如果音频有压缩，需要知道它的码率，音频流文件大小的计算方式如下：

$$\text{audio_size} = \text{bitrate} \times \text{time_in_seconds} / 8$$

例如，计算一个 10 分钟视频剪辑文件的最终大小，其中视频码率和音频码率分别为 1500kb/s 和 128kb/s，使用如下公式：

$$\text{file_size} = \text{video_size} + \text{audio_size}$$

$$\text{file_size} = (\text{video_bitrate} + \text{audio_bitrate}) \times \text{time_in_seconds} / 8$$

$$\text{file_size} = (1500\text{kb/s} + 128\text{kb/s}) \times 600\text{ s}$$

$$\text{file_size} = 1628\text{ kb/s} \times 600\text{ s}$$

$$\text{file_size} = 976800\text{ kb} = 976800000\text{ b} / 8 = 122100000\text{ B} / 1024 = 119238.28125\text{ KB}$$

$$\text{file_size} = 119238.28125\text{ KB} / 1024 = 116.443634033203125\text{ MB} \approx 116.44\text{ MB}$$

1 byte (B) = 8 bits (b)

1 kilobyte (KB) = 1024 B

1 megabyte (MB) = 1024 KB

7.5.2 调整视频分辨率

调整视频分辨率，主要是改变视频的宽和高。

可以通过在输出文件名之前设置-s选项来设置输出视频的宽度和高度。分辨率以 $w \times h$ 的形式输入，其中 w 是像素的宽度， h 是像素的高度。例如，为了调整从初始分辨率到 320×240 分辨率大小，使用如下命令：

```
ffmpeg -i input_file -s 320x240 output_file
```

若 FFmpeg 工具没有输入视频宽度和高度的确切数字，而是提供了预定义的列表表示对应的分辨率，那么接下来的两个命令有相同的结果，定义的 vga 对应 640×480 ：

```
ffmpeg -i input.avi -s 640x480 output.avi
```

```
ffmpeg -i input.avi -s vga output.avi
```

预定义的列表对应的分辨率如图 7-43 所示。

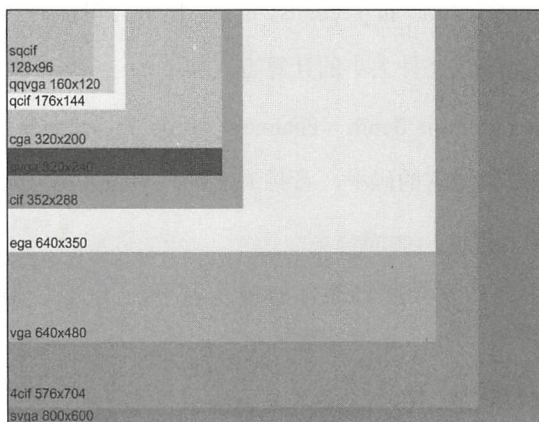


图 7-43 预定义的列表对应的分辨率

使用 `scale filter` 也可以替代 `-s` 选项来改变视频帧大小，如下命令可实现相同的效果：

```
ffmpeg -i input.mpg -s 320x240 output.mp4
ffmpeg -i input.mpg -vf scale=320:240 output.mp4
```

7.5.3 裁剪/填充视频

要裁剪视频，就意味着要选择好矩形区域，裁剪后只有矩形区域，没有其他部分。裁剪通常用于调整大小、填充和其他编辑操作。裁剪参数示意图如图 7-44 所示。

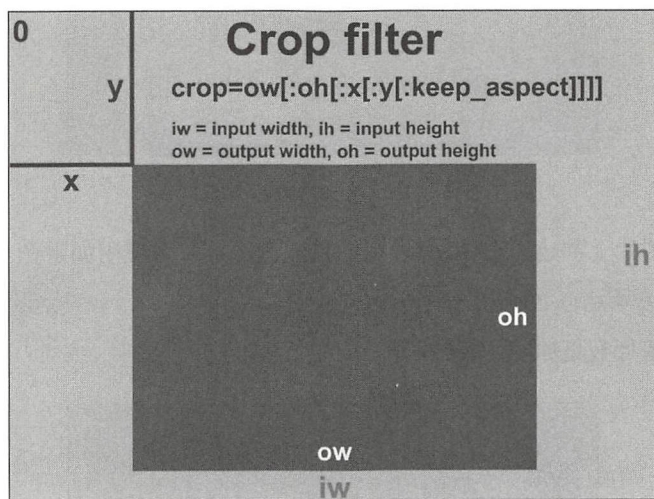


图 7-44 裁剪参数示意图

例如，矩形框靠左 1/3、靠中 1/3、靠右 1/3，使用如下命令：

```
ffmpeg -i input -vf crop=iw/3:ih:0:0 output
ffmpeg -i input -vf crop=iw/3:ih:iw/3:0 output
ffmpeg -i input -vf crop=iw/3:ih:iw/3*2:0 output
```

当要裁剪视频帧区域时，`crop filter` 的设计能够跳过 `x` 和 `y` 参数。默认的 `x` 及 `y` 的值如下：

```
xdefault = ( input width - output width ) / 2
ydefault = ( input height - output height ) / 2
```

默认从输入视频的中间区域开始，如下命令用于裁剪矩形中央 `w` 宽和 `h` 高的区域：

```
ffmpeg -i input_file -vf crop=w:h output_file
```

裁剪中间一半：

```
ffmpeg -i input.avi -vf crop=iw/2:ih/2 output.avi
```

下面介绍填充视频的内容。

在视频中添加一个额外的区域来添加额外的内容，如图 7-45 所示。

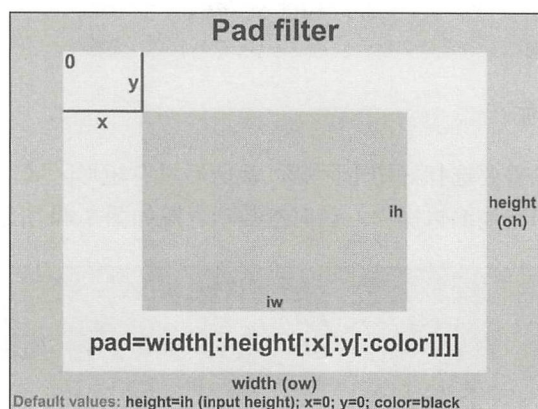


图 7-45 填充内容示意图

例如，在一张图片上创建一个 30 像素宽的粉红色边框，可使用如下命令：

```
ffmpeg -i photo.jpg -vf pad=860:660:30:30:pink framed_photo.jpg
```

运行命令后的图片效果如图 7-46 所示。

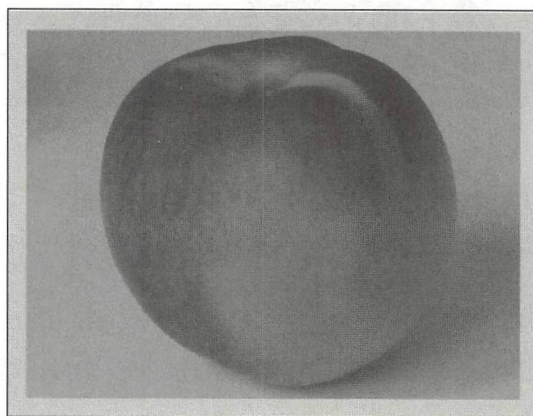


图 7-46 给图片加上 30 像素宽的粉红色边框

下面介绍如何将 4:3 的视频填充到 16:9 的视频。

有些设备只能在 16:9 的宽高比下播放视频，这时候视频需要在宽高对应的方向上填充。在这种情况下，高度保持不变，宽度等于高度值乘以 16/9。 x 值（输入视频帧水平偏移量）用表达式 $(\text{outputwidth} - \text{inputwidth})/2$ 计算，使用如下命令：

```
ffmpeg -i input -vf pad=ih*16/9:ih:(ow-iw)/2:0:color output
```

当不知道视频文件确切的分辨率时，默认填充的颜色就是黑色，命令如下：

```
ffmpeg -i film.mpg -vf pad=ih*16/9:ih:(ow-ih)/2:0 film_wide.avi
```

将 4:3 的视频填充到 16:9 的视频后的效果如图 7-47 所示。

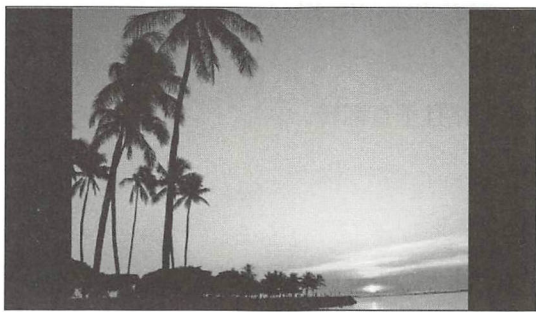


图 7-47 将 4:3 的视频填充到 16:9 的视频后的效果

当从 16:9 的视频转化成 4:3 的视频时，需要保持宽度不变，高=宽 \times 3/4，命令如下：

```
ffmpeg -i input -vf pad=iw:iw*3/4:0:(oh-ih)/2:color output
```

当不知道视频文件确切的分辨率时，使用如下命令：

```
ffmpeg -i hd_video.avi -vf pad=iw:iw*3/4:0:(oh-ih)/2 video.avi
```

从 16:9 的视频转化成 4:3 的视频后的效果如图 7-48 所示。

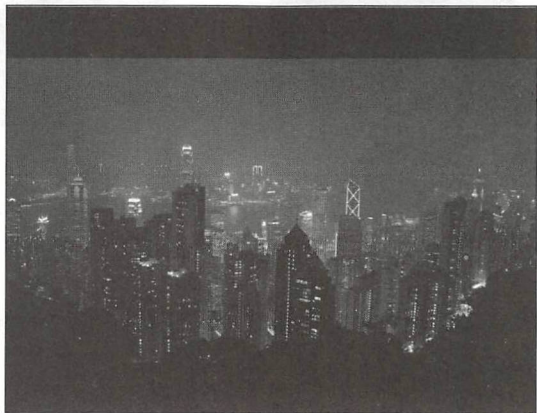


图 7-48 从 16:9 的视频转化成 4:3 的视频后的效果

7.5.4 翻转和旋转视频

视频帧的翻转和旋转是常见的视觉操作，可以用来创建各种各样的图形。

水平翻转，使用参数 `-vf hflip`，命令如下：

```
ffplay -f lavfi -i testsrc -vf hflip
```

垂直翻转，使用参数 `-vf vflip`，命令如下：

```
ffplay -f lavfi -i testsrc -vf vflip
```

旋转，使用参数 `transpose`，其值可以取下列值。

0：逆时针方向旋转 90° 并且垂直翻转。

1：顺时针方向旋转 90° 。

2：逆时针方向旋转 90° 。

3：顺时针方向旋转 90° 并且垂直翻转。

逆时针方向旋转 90° 并且垂直翻转，命令如下：

```
ffmpeg -i CMYK.avi -vf transpose=0 CMYK_transposed.avi
```

旋转后的效果如图 7-49 所示。

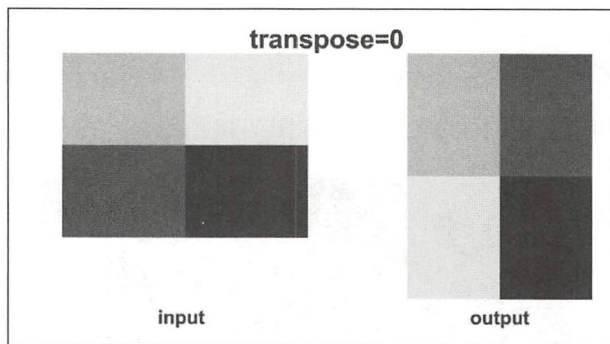


图 7-49 逆时针方向旋转 90° 并且垂直翻转后的效果

顺时针方向旋转 90° ，命令如下：

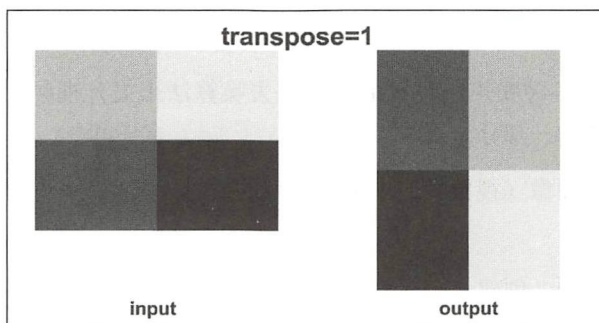
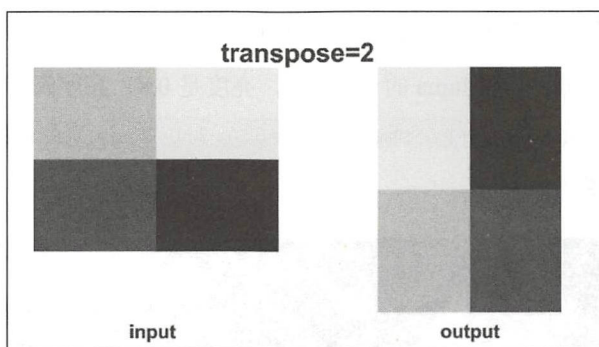
```
ffmpeg -i CMYK.avi -vf transpose=1 CMYK_transposed.avi
```

旋转后的效果如图 7-50 所示。

逆时针方向旋转 90° ，命令如下：

```
ffmpeg -i CMYK.avi -vf transpose=2 CMYK_transposed.avi
```

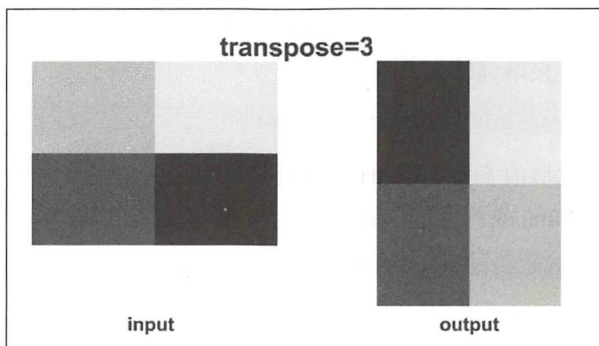
旋转后的效果如图 7-51 所示。

图 7-50 顺时针方向旋转 90° 后的效果图 7-51 逆时针方向旋转 90° 后的效果

顺时针方向旋转 90° 并且垂直翻转，命令如下：

```
ffmpeg -i CMYK.avi -vf transpose=3 CMYK_transposed.avi
```

旋转后的效果如图 7-52 所示。

图 7-52 顺时针方向旋转 90° 并且垂直翻转后的效果

7.5.5 模糊和锐化视频

输入视频一般包含各种噪声，可以通过一些去噪算法来提升视频的质量。在视频被编码前，去噪是视频预处理的一部分。

模糊效果用于提高图像（视频帧）中某些类型的质量，其中每个输出像素值都是从相邻的像素值计算出来的。

例如，输入一个视频，创建一个模糊效果。在这个视频中，luma 的半径是 1.5，luma 的权值是 1，代码如下：

```
ffmpeg -i input.mpg -vf boxblur=1.5:1 output.mp4
```

当我们不想影响图像的轮廓时，可以使用 smartblur 滤波器来进行处理。

为改善半色调图，可以设置 luma 的半径是 1，亮度是 0.8，亮度阈值是 0，命令如下：

```
ffmpeg -i halftone.jpg -vf smartblur=5:0.8:0 blurred_halftone.png
```

操作后的效果如图 7-53 所示。



图 7-53 使用 smartblur 滤波器后的效果

锐化或模糊视频帧可以使用 unsharp 滤波器。锐化滤波器可以用于普通非锐化模糊及高斯模糊。例如，使用默认值来增强输入，命令如下：

```
ffmpeg -i input -vf unsharp output.mp4
```

输出将会是一个锐化后的 5×5 的 luma 矩阵且强度为 1.0，为了创建一个高斯模糊效果，我们可以用一个负数来做 luma 或色度值，例如：

```
ffmpeg -i input -vf unsharp=6:6:-2 output.mp4
```

视频滤波器 denoise3d 用于去噪，是 MP Player 中滤波器的一部分。例如，使用 denoise3d 滤波器的默认值，可以增强去噪：

```
ffmpeg -i input.mpg -vf mp=denoise3d output.webm
```


图 7-54 展示了美国宇航局阿波罗项目使用 denoise3d 过滤器默认值的增强型存档视频。

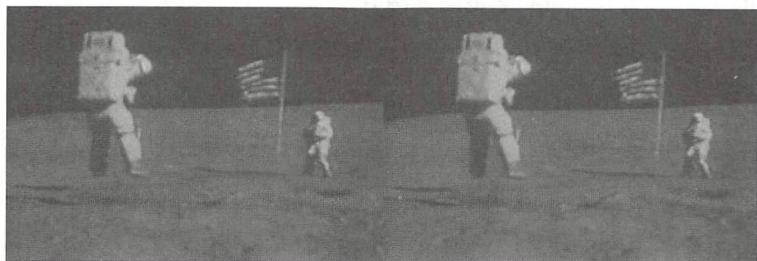


图 7-54 使用 denoise3d 过滤器默认值的增强型存档视频

可以用 hqdn3d 过滤器默认值来降低输入视频的噪声，命令如下：

```
ffmpeg -i input.avi -vf hqdn3d output.mp4
```

图 7-55 说明了 hqdn3d 过滤器的使用方法，不同的值对应的效果不同。

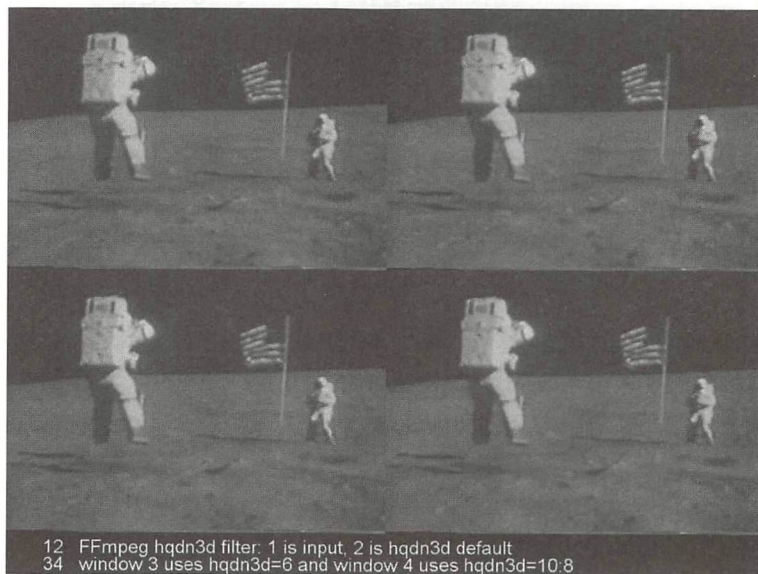


图 7-55 使用 hqdn3d 过滤器不同值对应的效果不同

7.5.6 画中画

视频叠加在日常生活中经常被用到，如电视台右上角的电视台 Logo，还有就是画中画效果，小窗口包含指定的内容，如央视新闻频道右下方有手语画面，另外就是一些广告浮层、滚动文字通知等。

Android 音视频开发

视频叠加是一种技术，通常用于在背景视频或图像上显示前景视频或图像。视频浮层叠加命令如下（input1 是视频背景，input2 是前景图）：

```
ffmpeg -i input.avi -vf hqdn3d output.mp4
```

需要注意的是，可以使用-vf选项来代替-filter_complex选项，因为现在有两个输入源：

```
ffmpeg -i input1 -vf movie=input2[logo];[in][logo]overlay=x:y output
```

另一种方法是 将一个输入分成几个输出，使用 pad 滤波器创建一个更大的背景图。在 filterchain 中使用此背景作为覆盖过滤器的第一个输入。背景覆盖背景图如图 7-56 所示。

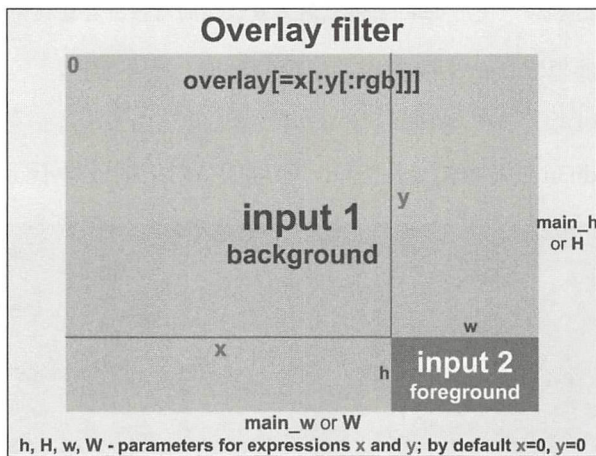


图 7-56 背景覆盖背景图

为了让主视频区域可见，Logo 经常放在屏幕的 4 个角落，接下来将有 4 个例子分别展示 4 个不同位置。视频分辨率大小是 1280×720，Logo 的大小是 150×140，标示的左上角（x 和 y 坐标）的正确位置来自背景和前景的宽度和高度值。

W, H ——背景图的宽和高。

w, h ——前景图（Logo）的宽和高。

Logo 在左上角，命令如下：

```
ffmpeg -i pair.mp4 -i logo.png -filter_complex overlay pair1.mp4
```

加 Logo 的效果如图 7-57 所示。

Logo 在右上角，命令如下：

```
ffmpeg -i pair.mp4 -i logo.png -filter_complex overlay=W-w pair2.mp4
```

加 Logo 的效果如图 7-58 所示。



图 7-57 左上角加上 Logo 图标

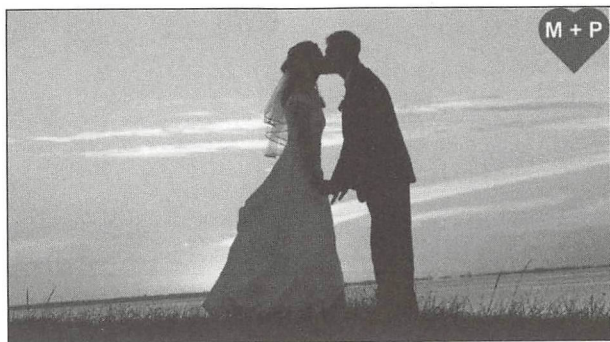


图 7-58 右上角加上 Logo 图标

Logo 在右下角，命令如下：

```
ffmpeg -i pair.mp4 -i logo.png -filter_complex overlay=W-w:H-h pair3.mp4
```

加 Logo 的效果如图 7-59 所示。



图 7-59 右下角加上 Logo 图标

Android 音视频开发

Logo 在左下角，命令如下：

```
ffmpeg -i pair.mp4 -i logo.png -filter_complex overlay=0:H-h pair4.mp4
```

下面我们再介绍一下让 Logo 在指定的时刻显示的方法。

当视频需要一些特殊的介绍时，Logo（或其他的视频浮层）可以在一定时间间隔后添加，如 5s 后，蓝色背景中包含一个红色 Logo，使用如下命令：

```
ffmpeg -i video_with_timer.mp4 -itsoffset 5 -i logo.png ^  
-filter_complex overlay timer_with_logo.mp4
```

在指定时刻显示 Logo 的效果如图 7-60 所示。

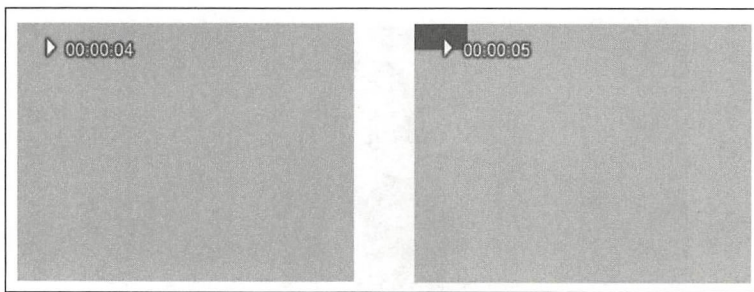


图 7-60 定时展示 Logo 图标

然后介绍一下在视频中加入计时器的方法。

下面的例子是 1973 年 NASA 提供的一个视频，其中是阿波罗 17 号在月球表面的轨道，视频持续 29.93s，分辨率大小是 512×384，使用两位数的计时器，命令如下：

```
ffmpeg -f lavfi -i testsrc -vf crop=61:52:224:94 -t 30 timer.ogg
```

现在在视频区域有一个 61×52 像素大小显示计时器的区域，时间范围为 0~30s，视频将为阿波罗 17 号登月的过程计时，使用如下命令：

```
ffmpeg -i start.mp4 -i timer.ogg -filter_complex overlay=451 start1.mp4
```

计时器的 x 坐标是 $512-61=451$ ，y 坐标是 0。该例子的效果如图 7-61 所示。

将计时器缩小到原来的 1/2，并将其放置在底部，使用如下命令：

```
ffmpeg -i start.mp4 -vf movie=timer.ogg,scale=15:14[tm];  
[in][tm]overlay=248:371 overlay.mp4
```

修改后的计时器效果如图 7-62 所示。

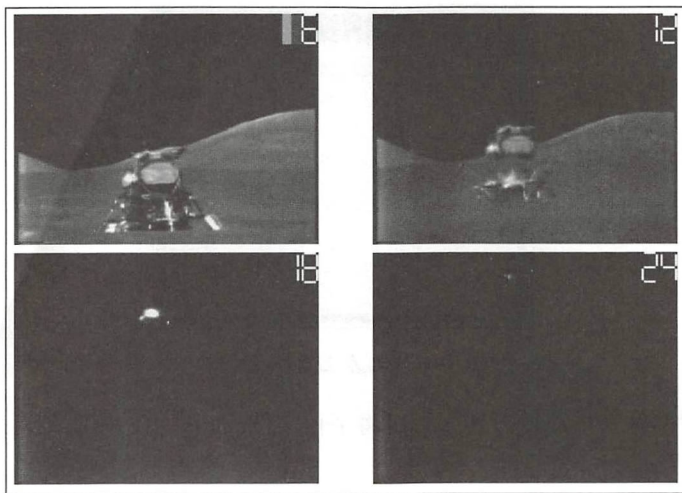


图 7-61 视频中展示计时器

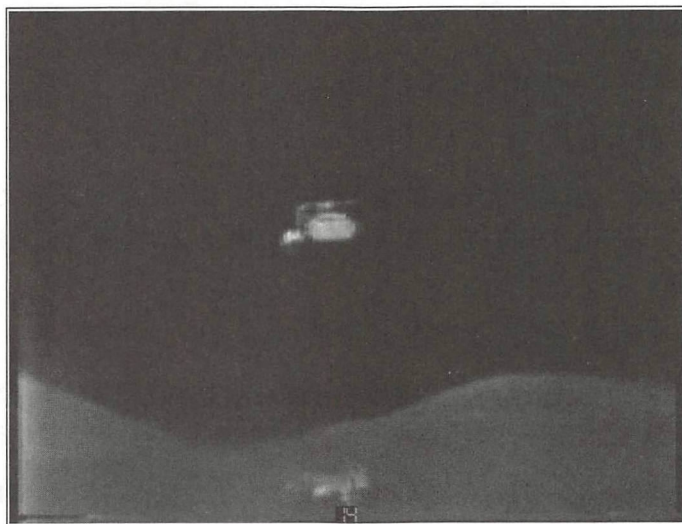


图 7-62 视频中指定位置展示缩小的计时器

7.5.7 在视频上添加文字

视频中包含文本数据可以极大地提高信息质量。在视频中添加文字，有两种方案：一种是在视频输出时，将文字包含在其中，相当于叠加在上面；另外一种是使用 `drawtext filter` 来处理。例如，要绘制一条欢迎消息（默认位于左上角），并使用 Arial 字体，字体颜色为黑色，如图 7-63 所示。

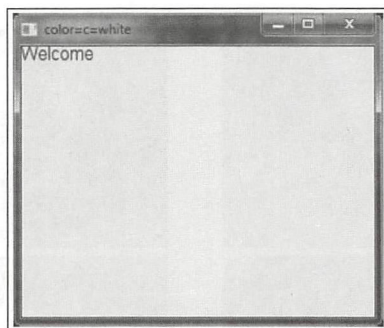


图 7-63 在左上角绘制消息

如果无法指定字体文件路径，那么可以将字体文件（arial.ttf）复制到当前目录下，可以使用如下命令：

```
ffmpeg -f lavfi -i color=c:white -vf drawtext=fontfile=arial.ttf:text=Welcome
```

1. 文字位置的设置

文字位置可以通过 x 和 y 参数值指定具体的值。

2. 水平方向上的设置

通过将 x 坐标设置为需要的值来处理水平文本位置，例如， $x=40$ 表示文本距离左边界 40 像素； $x=(w-tw)/2$ 表示将文本定位到水平中心（其中， tw 为文本宽度， w 为帧宽）； $x=w-tw$ 表示将文本对齐到右边。

3. 垂直方向上的设置

y 坐标的设置决定了垂直文本位置，例如， $y=50$ 表示文本距离顶端 50 像素， $y=(h-th)/2$ 表示将文本定位到垂直中心（其中， th 为文本高度， h 为帧高）； $y=h$ 表示将文本对齐到底端。

例如，将文本放在视频帧的中心位置。如果文本中包含空格，当使用 FFmpeg 输入文字时，可以使用专门的滤镜，名为 `drawtext`（注意其中若有空格，将包含在成对单引号或成对双引号中），代码如下：

```
ffmpeg -f lavfi -i color=c:white -vf drawtext="fontfile=arial.ttf:text='Good day':x=(w-tw)/2:y=(h-th)/2"
```

在视频帧的中心位置绘制文本的效果如图 7-64 所示。

4. 字体大小和颜色设置

例如，使用字体大小为 30，颜色为绿色的文字，代码如下：


```
ffplay -f lavfi -i color=c=white -vf drawtext="fontfile=arial.ttf:
text='Happy Holidays':x=(w-tw)/2:y=(h-th)/2:
fontcolor=green:fontsize=30"
```

设置字体大小和颜色后的效果如图 7-65 所示。

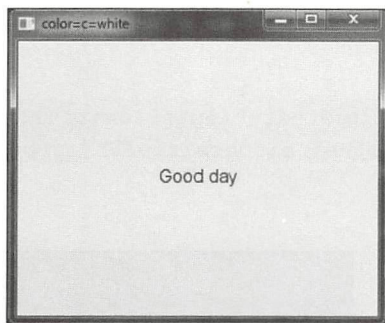


图 7-64 在视频帧的中心位置绘制文本的效果

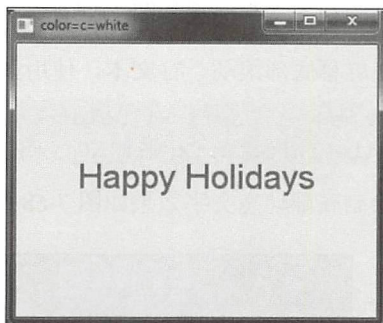


图 7-65 设置字体大小和颜色后的效果

当我们想要将背景色改为蓝色，字体颜色改为黄色时，使用如下命令：

```
ffplay -f lavfi -i color=c=blue -vf drawtext=^
"fontfile=arial.ttf:text='Happy Holidays':x=(w-tw)/2:y=(h-th)/2:
fontcolor=yellow:fontsize=40"
```

更改背景色和字体颜色后的效果如图 7-66 所示。

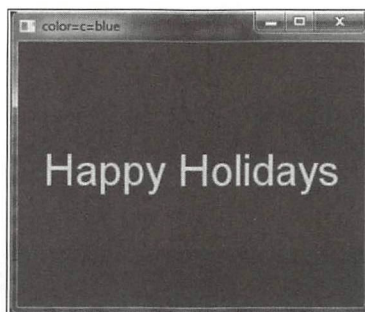


图 7-66 更改背景色和字体颜色后的效果

5. 动态文字设置

(1) 水平方向上文字的运动

为了在视频帧中水平移动文本，我们将 t 变量（表示时间，单位为 s ）包含到 x 的表达式中。例如，我们想要在从右向左的方向上每秒移动一个文本，每次移动 n 个像素，使用的表达式为 $x = w - t \times n$ 。为了使运动在从左向右的方向上改变，使用表达式 $x = w + t$ 。例如，为了在屏

Android 音视频开发

幕顶部移动“mic RTL text”字符串，使用如下命令：

```
ffmpeg -f lavfi -i color=c=#abcdef -vf drawtext=^
"fontfile=arial.ttf:text='Dynamic RTL text':x=w-t*50:^
fontcolor=darkorange:fontsize=30" output
```

绘制动态文字效果如图 7-67 所示。

在屏幕底部滚动一行文本，使用如下命令：

```
ffmpeg -f lavfi -i color=c=orange -vf drawtext="fontfile=arial.ttf:
textfile=info.txt:x=w-t*50:y=h-th:fontcolor=blue:fontsize=30" output
```

绘制底部动态文字效果如图 7-68 所示。

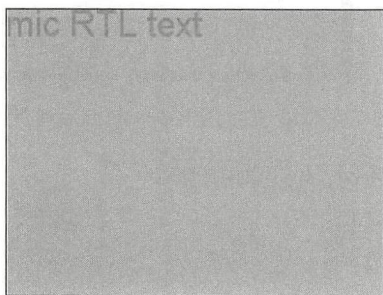


图 7-67 绘制动态文字效果

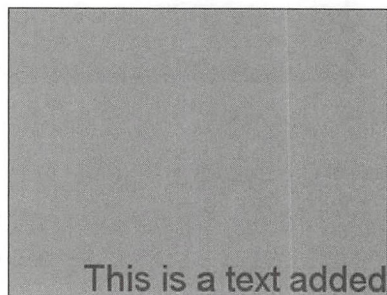


图 7-68 绘制底部动态文字效果

(2) 垂直文字滚动

垂直文字滚动，使用如下命令：

```
ffmpeg -i palms.avi -vf drawtext="fontfile=arial.ttf:textfile=Credits:^
x=(w-tw)/2:y=h-t*100:fontcolor=white:fontsize=30" clip.mp4
```

其设置效果如图 7-69 所示。

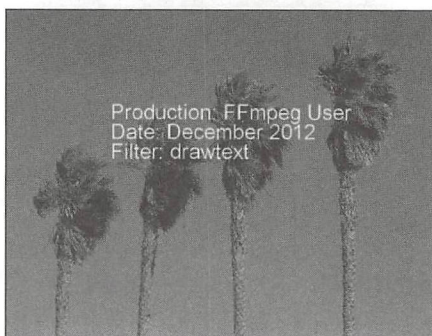


图 7-69 垂直文字滚动

7.5.8 文件格式转换

FFmpeg 工具经常与音频或视频格式的转换有关。另外，format 参数是在输出文件之前由 -f 选项设置的，也可以在 FFmpeg 输入参数时进行设置。下面介绍一下媒体格式文件类型，如图 7-70 所示。

Characteristics of common media containers										
Container	Support for particular file format									
	Audio					Video				
	AAC	AC-3	MP3	PCM	WMA	MPEG1 MPEG2	MPEG4	H.264/ MPEG4 AVC	VC1 WMV	Theora
AVI	Y	Y	Y	Y	Y	Y	Y	partially	Y	Y
Matroska	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
MP4	Y	Y	Y	N	Y	Y	Y	Y	Y	N
MXF	Y	Y	Y	Y	N	Y	Y	Y	Y	N
Ogg/OGM	N	N	Y	Y	N	Y	Y	Y	Y	Y
QuickTime	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

图 7-70 媒体格式文件类型

诸如 AVI、MP4 这些就是封装格式，AAC、H.264 这些就是编码格式。

FFmpeg 可以对输入的文件流进行文件格式的转换，比如只修改输入文件某些文件信息，就可以让输出文件的媒体格式保持和原来一样。输入文件的数据包在编码时进行压缩或不压缩，视频压缩包括使用特定的编解码器。文件格式转换过程可以分为如下几个步骤。

(1) 解复用过程 (demuxing)：将编码好的数据，通过 FFmpeg 的 libavformat 库和解复用器进行解复用，分离出原始音频和视频数据。

(2) 解码过程 (decoding)：将解复用后的音频和视频数据包 (AVPacket)，通过相匹配的解码器解码成未压缩的视频帧 (AVFrame)，如果 FFmpeg 命令中有 -c copy (或 -codec copy) 选项，则表示没有解码 (decoding) (和做图像滤镜 (filtering)) 过程发生。

(3) 编码过程 (encoding)：编码器可以把未压缩的帧 (AVFrame) 编码成原始数据包 (AVPacket)。

(4) 复用过程 (muxing)：通过复用器将原始数据包 (AVPacket) 合成指定的封装格式。

媒体格式和编解码器转换如图 7-71 所示。

Android 音视频开发

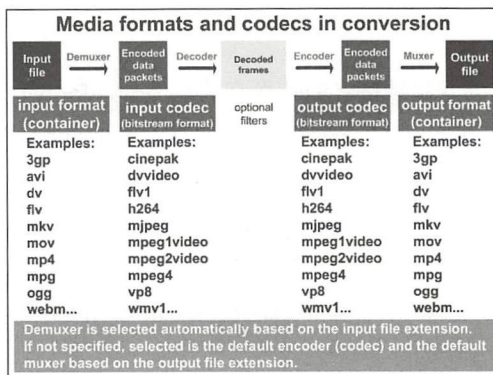


图 7-71 媒体格式和编解码器转换

下面介绍一下 Codec。

Codec 的名称来源于单词编码-解码器，表示一个设备或用于编码和解码视频流或音频流，或用于某种算法以进行压缩。

FFmpeg 编解码器定义的是一种媒体比特流格式。使用如下命令可以查看 FFmpeg 中可用的 Codec。

- `ffmpeg -codecs`: 展示所有编解码器。
- `ffmpeg -decoders`: 展示所有解码器。
- `ffmpeg -encoders`: 展示所有编码器。

默认编解码器对应的文件扩展类型如图 7-72 所示。

Default codecs (encoders) for common video file extensions (file formats)		
extension	codec	Additional data
.avi	mpeg4	mpeg4 (Simple profile), yuv420p; audio: mp3
.flv	flv1	yuv420p; audio: mp3
.mkv	h264	h264 (High), yuvj420p; audio: vorbis codec, fltp sample format
.mov	h264	h264 (High), yuvj420p; audio: aac (mp4a)
.mp4	h264	h264 (High), yuvj420p; audio: aac (mp4a)
.mpg	mpeg1video	yuv420p; audio: mp2
.ogg	theora	yuv422p, bitrate very low; audio excluded during conversion
.ts	mpeg2video	yuv422p; audio: mp2
.webm	vp8	yuv420p; audio: vorbis codec, fltp sample format
Default codecs (encoders) for common audio file extensions (file formats)		
extension	codec	Additional data
.aac	aac	libvo_aacenc, bitrate 128 kb/s
.flac	flac	FLAC (Free Lossless Audio Codec), bitrate 128 kb/s
.m4a	aac	mp4a, bitrate 128 kb/s
.mp2	mp2	MPEG Audio Layer 2, bitrate 128 kb/s
.mp3	mp3	libmp3lame, bitrate 128 kb/s
.wav	pcm_s16le	PCM (Pulse Code Modulation), uncompressed
.wma	wmav2	Windows Media Audio

图 7-72 默认编解码器对应的文件扩展类型

重写相同命名的输出文件:

```
ffmpeg -y -i input.avi output.mp4
```

其中, -y 选项表示强制重写。

7.5.9 时间操作

多媒体处理包含改变输入 duration、设置延迟。时间有如下两种格式:

```
[-]HH:MM:SS[.m...]
```

```
[-]S+[.m...]
```

下面介绍音频和视频时长设置。

可以通过-t 选项, 例如设置一个 MP3 文件的播放时长为 3 分钟, 使用如下命令:

```
ffmpeg -i music.mp3 -t 180 music_3_minutes.mp3
```

设置帧数的命令如下:

```
ffmpeg -i music.mp3 -t 180 music_3_minutes.mp3
```

```
audio: -aframes number or -frames:a number
```

```
data: -dframes number or -frames:d number
```

```
video: -vframes number or -frames:v number
```

设置一个 10 分钟 25fps 的视频, 可使用如下命令:

```
ffmpeg -i video.avi -vframes 15000 video_10_minutes.avi
```

设置延迟转换, 例如, 从第 10s 开始进行转换, 使用如下命令:

```
ffmpeg -i input.avi -ss 10 output.mp4
```

提取媒体文件中的部分内容, 例如, 提取某视频第 5 分钟的内容, 使用如下命令:

```
ffmpeg -i video.mpg -ss 240 -t 60 clip_5th_minute.mpg
```

第 8 章

FFmpeg 源码分析及实战

第 7 章主要介绍了 FFmpeg 框架，包括其在不同平台下的编译，以及通过 FFmpeg 的具体命令处理一些音视频效果，而本章主要介绍 FFmpeg 源码分析及一些实战案例。

8.1 FFmpeg 常用结构体分析

在分析 FFmpeg 常用结构体之前，我们先找到 Ffplay.c 文件，从其 main 函数入手，分析基本流程，看看需要介绍哪些结构体。

(1) void av_register_all(void)中没有相关结构体。

(2) int avformat_open_input(AVFormatContext **ps, const char *filename, AVInputFormat *fmt, AVDictionary **options)，这个函数将要分析 AVFormatContext 及 AVInputFormat，而 AVDictionary 结构较简单，暂时不做分析。

(3) int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options)，这个函数中涉及如下 AV 结构体：

```
AVStream *st;
AVCodecContext *avctx;
AVPacket pkt1, *pkt;
AVCodec *codec;
AVDictionary *thread_opt = NULL;
```

(4) avcodec_find_decoder 中涉及 AVFrame *frame = av_frame_alloc();。

(5) int attribute_align_arg avcodec_open2(AVCodecContext *avctx, const AVCodec *codec,

AVDictionary **options)中的结构体, 前面章节中已经介绍过, 此处不再重复介绍。

(6) int av_read_frame(AVFormatContext *s, AVPacket *pkt)中的结构体, 前面章节中已经介绍过, 此处不再重复介绍。

(7) int attribute_align_arg avcodec_decode_video2(AVCodecContext *avctx, AVFrame *picture, int *got_picture_ptr, const AVPacket *avpkt)中的结构体, 前面章节中已经介绍过, 此处不再重复介绍。

(8) int attribute_align_arg avcodec_decode_audio4(AVCodecContext *avctx, AVFrame *frame, int *got_frame_ptr, const AVPacket *avpkt)中的结构体, 前面章节中已经介绍过, 此处不再重复介绍。

本章后续章节将按照先后顺序, 分析出现的结构体。

8.1.1 AVFormatContext

AVFormatContext 是一个贯穿全局的数据结构, 很多函数都要用它作为参数。FFmpeg 代码中对这个数据结构的注释是 format I/O context, 此结构包含了一个视频流的格式内容。其中存有 AVInputFormat (或 AVOutputFormat, 但是同一时间 AVFormatContext 内只能存在其中一个)、AVStream、AVPacket 这几个重要的数据结构以及一些其他的相关信息, 比如 title、author、copyright 等。另外, 还有一些可能在编解码中会用到的信息, 诸如 duration、file_size、bit_rate 等。

由于 AVFormatContext 结构包含许多信息, 因此初始化过程是分步完成的, 而且其中有些变量如果没有值可用, 也可不初始化。但是由于一般声明都使用指针, 因此分配内存的过程不可少。如下代码初始化了 AVFormatContext:

```
AVFormatContext *pFormatCtx;
pFormatCtx = avformat_alloc_context();
```

结构体定义如下:

```
typedef struct AVFormatContext {
    const AVClass *av_class; // 与 logging 及 avoptions 相关的 class
    struct AVInputFormat *iformat; // 输入容器格式
    struct AVOutputFormat *oformat; // 输出容器格式
    void *priv_data; // 私有数据
    AVIOContext *pb; // I/O Context
    unsigned int nb_streams; // 流的数目
    AVStream **streams;
    // 文件中的 Stream 列表, 可以通过 avformat_new_stream 创建新的 Stream
    char filename[1024]; // 输入或输出文件名
```

```

int64_t duration;//Stream 的时长
int64_t bit_rate;//整个流的比特率
unsigned int packet_size;//Packet 的大小
int max_delay;
int flags;//Demuxer/Muxer 的状态
//此处省略部分宏定义
int audio_preload;//音频预加载
#define AVFMT_FLAG_FAST_SEEK 0x80000 //快速 seek, 部分格式的 seek 精度可能不准确
int64_t probesize;//读取数据大小 (要足够起播首帧画面)
int64_t max_analyze_duration;//通过 avformat_find_stream_info 读取最大时长
const uint8_t *key;
int keylen;
unsigned int nb_programs;
AVProgram **programs;
enum AVCodecID video_codec_id;//视频的 codec_id
enum AVCodecID audio_codec_id;//音频的 codec_id
enum AVCodecID subtitle_codec_id;//字幕的 codec_id
unsigned int max_index_size;//每条流的最大内存字节数
unsigned int max_picture_buffer;//Buffering Frame 中最大的内存字节数
unsigned int nb_chapters;
AVChapter **chapters;
AVDictionary *metadata;//整个 metadata 的源数据
int64_t start_time_realtime;//起始时间, 从 PTS=0 开始
int fps_probe_size;//在 avformat_find_stream_info 中, 用于确定帧数
int error_recognition;//错误检测
AVIOInterruptCB interrupt_callback;//自定义的中断 callback
int debug;//标记是否 debugging
#define FF_FDEBUG_TS 0x0001
int64_t max_interleave_delta;
//最大交叉 Buffering (缓冲数据) 时长, 在 muxing (复用) 时使用
int strict_std_compliance;//允许非标准扩展
int event_flags;//时间 flag
#define AVFMT_EVENT_FLAG_METADATA_UPDATED 0x0001
int max_ts_probe;//解码第 1 帧时读取的最大 Packet 数目
int avoid_negative_ts;//在 muxing (复用) 时, 避免无效时间戳
#define AVFMT_AVOID_NEG_TS_AUTO -1
#define AVFMT_AVOID_NEG_TS_MAKE_NON_NEGATIVE 1
#define AVFMT_AVOID_NEG_TS_MAKE_ZERO 2
int ts_id;//ts 流的 id
int max_chunk_duration;//最大 chunk 时长, 不是所有的格式都支持 chunk 方式
int max_chunk_size;//最大 chunk 以字节为单位。注意, 不是所有的格式都支持 chunk 方式
int use_wallclock_as_timestamps;
//强制使用 wallclock 时间戳作为数据包的 PTS/DTS, B 帧存在时, 未定义
int avio_flags;
enum AVDurationEstimationMethod duration_estimation_method;
//可以通过不同的方法估计持续时间字段

```



```

int64_t skip_initial_bytes; //在打开流时, 跳过初始字节
unsigned int correct_ts_overflow; //纠正单个时间戳溢出
int seek2any; //强制快进/快退到任意帧
int flush_packets; //在每个 packet 后, 刷新 I/O Context
int probe_score; //格式探测评分, 最大评分是 AVPROBE_SCORE_MAX
int format_probesize; //读取最大的字节数以确定格式
char *codec_whitelist; //所有可用的 Decoder
char *format_whitelist; //所有可用的 Demuxer
AVFormatInternal *internal; //libavformat 内部私有成员
int io_repositioned; //I/O 更改的标志
AVCodec *video_codec; //特殊解码器或相同 codec_id 的视频 Codec
AVCodec *audio_codec; //特殊解码器或相同 codec_id 的音频 Codec
AVCodec *subtitle_codec; //特殊解码器或相同 codec_id 的字幕 Codec
AVCodec *data_codec; //特殊解码器或相同 codec_id 的数据 Codec
int metadata_header_padding; //在 metadata 头设置的 padding 值
void *opaque; //私有数据
av_format_control_message control_message_cb; //设备和应用通信的 callback
int64_t output_ts_offset; //输出时间戳偏移值
uint8_t *dump_separator;
enum AVCodecID data_codec_id; //数据 codec_id
int (*open_cb)(struct AVFormatContext *s, AVIOContext **p, const char
*url, int flags, const AVIOInterruptCB *int_cb, AVDictionary **options);
//过时函数, 使用 io_open_and_io_close 代替
char *protocol_whitelist; //协议白名单
int (*io_open)(struct AVFormatContext *s, AVIOContext **pb, const char
*url, int flags, AVDictionary **options);
//当 I/O 流打开时, demux (解复用) 操作的回调函数
void (*io_close)(struct AVFormatContext *s, AVIOContext *pb);
//AVFormatContext 打开时的回调函数
char *protocol_blacklist; //协议黑名单
} AVFormatContext;

```

8.1.2 AVInputFormat

AVInputFormat 是 FFmpeg 的解复用器对象, AVInputFormat 是类似 COM 接口的数据结构, 表示输入文件容器格式, 着重于功能函数, 一种文件容器格式对应一个 AVInputFormat 结构, 在程序运行时有多实例。next 变量用于把支持的所有输入文件容器格式连接成链表, 便于遍历查找; priv_data_size 标示具体的文件容器格式对应的 Context 的大小, 对应的结构体定义如下:

```

typedef struct AVInputFormat {
    const char *name; //格式的短名称
    const char *long_name; //格式的长名称
    int flags;

```



```

const char *extensions;//定义了扩展, 不会进行格式探测, 通常不使用扩展
const struct AVCodecTag * const *codec_tag;//Codec 的 tag
const AVClass *priv_class; //AVClass 用于内部 Context
const char *mime_type;//mime 类型, 如 video/avc, 在 probing (探测) 时需要检查
struct AVInputFormat *next;
int raw_codec_id;//原始 Demuxer 存储的 codec id
int priv_data_size;//私有数据
int (*read_probe)(AVProbeData *);
//读取 probe 数据, 提供的 probe 数据须保证 AVPROBE_PADDING_SIZE
int (*read_header)(struct AVFormatContext *);
//读取 format 头并初始化 AVFormatContext 结构, 返回 0, 表示 OK
int (*read_packet)(struct AVFormatContext *, AVPacket *pkt);
//读取一个 packet 并且存入 pkt 指针中
int (*read_close)(struct AVFormatContext *);//关闭流
int (*read_seek)(struct AVFormatContext *,
int stream_index, int64_t timestamp, int flags);//读取给定的时间戳
int64_t (*read_timestamp)(struct AVFormatContext *s, int stream_index,
int64_t *pos, int64_t pos_limit);
//读取 stream[stream_index] 中的下一个时间戳, 如果出错, 返回 AV_NOPTS_VALUE
int (*read_play)(struct AVFormatContext *);
//恢复播放, 仅仅在网络协议 RTSP 下才有意义
int (*read_pause)(struct AVFormatContext *);
//暂停, 仅仅在网络协议 RTSP 下才有意义
int (*read_seek2)(struct AVFormatContext *s, int stream_index, int64_t min_ts,
int64_t ts, int64_t max_ts, int flags);//快进/快退到指定时间戳
int (*get_device_list)(struct AVFormatContext *s, struct AVDeviceInfoList
*device_list);//返回设备列表及其属性
int (*create_device_capabilities)(struct AVFormatContext *s, struct
AVDeviceCapabilitiesQuery *caps);//初始化设备能力子模块
int (*free_device_capabilities)(struct AVFormatContext *s, struct
AVDeviceCapabilitiesQuery *caps);//释放设备能力子模块
} AVInputFormat;

```

8.1.3 AVStream

AVStream 是存储每一个视频/音频流信息的结构体。首先看一下结构体的定义（位于 avformat.h 文件中）：

```

typedef struct AVStream {
int index; //AVFormatContext 流索引
int id;//stream ID
void *priv_data;//私有数据
AVRational time_base;//时间基, 以 s 为基本单位
int64_t start_time;//按显示图像排序的第 1 帧的 PTS, 注意, ASF Header 不包含正确的
//start_time, ASF Demuxer 不要设置这个属性

```

```

int64_t duration; //流的时长
int64_t nb_frames; //流中的帧数
int disposition; /**< AV_DISPOSITION_* bit field */
enum AVDiscard discard; //选择哪个 Packet 是废弃的, 不需要 demux (解复用)
AVRational sample_aspect_ratio; //采样率
AVDictionary *metadata; //元数据
AVRational avg_frame_rate; //平均帧率
AVPacket attached_pic;
//对于有 AV_DISPOSITION_ATTACHED_PIC 的配置, packet 将包含附加图像
AVPacketSideData *side_data; //整个流的 side data
int nb_side_data; //side data 的个数
int event_flags; //检测流时的标示
#define AVSTREAM_EVENT_FLAG_METADATA_UPDATED 0x0001 //更新 metadata 结果标示
#define MAX_STD_TIMEBASES (30*12+30+3+6)
//av_find_stream_info 需要的流信息
struct {
    int64_t last_dts; //上一个 DTS
    int64_t duration_gcd;
    int duration_count;
    int64_t rfps_duration_sum;
    double (*duration_error)[2][MAX_STD_TIMEBASES];
    int64_t codec_info_duration;
    int64_t codec_info_duration_fields;
    int found_decoder; //大于 0, 表示找到; 小于或等于 0, 表示还没有找到
    int64_t last_duration;
    int64_t fps_first_dts;
    int fps_first_dts_idx;
    int64_t fps_last_dts;
    int fps_last_dts_idx;
} *info;

int pts_wrap_bits; /**< 用于装饰控制时表示 PTS 数目*/
int64_t first_dts; //第一个 DTS
int64_t cur_dts; //当前 DTS
int64_t last_IP_pts;
int last_IP_duration;
int probe_packets; //Codec 探测出用于 Buffer 的 packet 数
int codec_info_nb_frames; //demux (解复用) 后的帧数
/* av_read_frame() support */
enum AVStreamParseType need_parsing;
struct AVCodecParserContext *parser;
struct AVPacketList *last_in_packet_buffer;
//packet_buffer 中的上一个 Packet

```



```

AVProbeData probe_data;
#define MAX_REORDER_DELAY 16
int64_t pts_buffer[MAX_REORDER_DELAY+1];
AVRational r_frame_rate;//帧率
int stream_identifier;//stream 识别, 0 表示未知
int request_probe;//stream 探测状态, -1 表示探测完成, 0 表示没有探测请求
int skip_to_keyframe;//跳过下一个关键帧
int skip_samples;//帧开始时, 下一个 Packet 中解码的样本数
int64_t start_skip_samples;
int64_t first_discard_sample;//第一个废弃的采样
int64_t last_discard_sample;//上次废弃的采样, 避免 EOF
int nb_decoded_frames;//解码后的帧数
int64_t mux_ts_offset;//在 muxing (复用) 时, 给时间戳添加偏移量
int64_t pts_wrap_reference;//检查时间戳的包装
int pts_wrap_behavior;//PTS 的扩展行为
int update_initial_durations_done;//避免更新初始化时间两次
//从 PTS 生成 DTS
int64_t pts_reorder_error[MAX_REORDER_DELAY+1];
uint8_t pts_reorder_error_count[MAX_REORDER_DELAY+1];
int64_t last_dts_for_order_check;
uint8_t dts_ordered;
uint8_t dts_misordered;
int inject_global_side_data;//内部数据注入全局 side data
char *recommended_encoder_configuration;//推荐使用的键/值形式的编码器配置项
AVRational display_aspect_ratio;//显示率
struct FFFrac *priv_pts;
AVStreamInternal *internal;//libavformat 内部使用的成员
AVCodecParameters *codecpar;//Codec 参数, 通过 avformat_new_stream 分配, 通过
//avformat_free_context 释放
} AVStream;

```

解复用器的目的就是从容容器中分离（解析出来）不同的流，FFmpeg 中的流对象为 AVStream，它是由解复用器的 read_header 函数创建的，并保存在 AVFormatContext 的 nb_streams（容器中的流条数）及 streams 数组中。

以 ff_srt_demuxer 为例，它的 read_header 指向 srt_read_header 函数，该函数主要完成以下功能。

（1）调用 avformat_new_stream 函数创建一条流。

- 调整 AVFormatContext->streams 的大小。
- 申请流空间并清零。
- 申请 AVStream->info 空间并清零。

- 申请 AVStream->codec (AVCodecContext 对象) 空间并设置默认值 (由 avcodec_get_context_defaults3 函数设置)。
- 设置流的其余参数的默认值 (目前大部分的成员不知道作用, 因此先不关注)。
- 将流指针放入 AVFormatContext->streams 数组的末尾。

(2) 设置 AVStream->codec->codec_type 及 AVStream->codec->codec_id 分别为 AVMEDIA_TYPE_SUBTITLE、AV_CODEC_ID_SUBRIP。

(3) 循环读入每一条字幕 (ff_subtitles_read_chunk 函数), 并从中解析出字幕信息 (PTS、duration 及字幕内容), 然后调用 ff_subtitles_queue_insert 函数将字幕内容插入队列并返回 AVPacket 指针 (以便插入从字幕中解析出的私有数据)。

(4) 读入完成后, 调用 ff_subtitles_queue_finalize 函数对队列中的数据进行排序, 并调整字幕持续时间 (如果有必要)。

8.1.4 AVCodecContext

这是一个描述编解码器上下文的数据结构, 包含了众多编解码器需要的参数信息, 如果是单纯地使用 libavcodec, 这部分信息需要调用者进行初始化; 如果是使用整个 FFmpeg 库, 这部分信息在调用 av_open_input_file 和 av_find_stream_info 的过程中会根据文件的头部信息及媒体流内的头部信息完成初始化。AVCodecContext 代码如下:

```
typedef struct AVCodecContext {
    const AVClass *av_class;
    //av_log 使用的结构体, 通过 avcodec_alloc_context3 设置
    int log_level_offset; //日志中的级别
    enum AVMediaType codec_type; /*参见 AVMEDIA_TYPE_xxx*/
    const struct AVCodec *codec;
    enum AVCodecID codec_id; /*参见 AV_CODEC_ID_xxx*/
    unsigned int codec_tag; //Codec 的标记
    void *priv_data;
    struct AVCodecInternal *internal; //内部使用的上下文环境
    void *opaque;
    int64_t bit_rate; //平均比特率
    int global_quality; //全局 Codec 质量
    int gop_size; //GOP 的大小
    int has_b_frames; //是否有 B 帧
    int slice_count; //slice 总和
    int *slice_offset; //slice 的偏移量
    AVRational sample_aspect_ratio; //采样率
    int slice_flags; //slice 标示
    int refs; //参考帧的数目
```

```

enum AVColorSpace colorspace; //YUV 颜色空间类型
enum AVColorRange color_range; //用于存储颜色范围
int slices; //表示图像细分的数量, 用于并行解码
/*以下用于音频相关操作*/
int sample_rate; //采样率
int channels; //信道数
enum AVSampleFormat sample_fmt; //音频采样格式
int frame_size; //音频每个信道的采样数
int frame_number; //帧计数器
uint64_t channel_layout; //音频信道布局
//获取 Buffer 的回调函数
int (*get_buffer2)(struct AVCodecContext *s, AVFrame *frame, int flags);
int refcounted_frames; //通过 avcodec_decode_video2 和 avcodec_decode_audio4
//返回的解码后的帧数
int rc_buffer_size; //解码器比特流的 Buffer 大小
struct AVHWAccel *hwaccel; //硬件加速结构体
void *hwaccel_context; //硬件加速 Context (上下文环境)
int thread_count; //线程个数, 取决于有多少个独立任务在执行
int profile;
int level;
enum AVDiscard skip_frame; //跳过选定帧的解码操作
uint8_t *subtitle_header; //字幕头
int subtitle_header_size; //字幕头大小
AVRational framerate; //压缩码流中存储的帧率
AVRational pkt_timebase; //pkt_dts 和 pkt_pts 的时间单元
int64_t pts_correction_num_faulty_pts; ///当前不正确 PTS 值的数目
int64_t pts_correction_num_faulty_dts; ///当前不正确 DTS 值的数目
int64_t pts_correction_last_pts; ///上一帧的 PTS
int64_t pts_correction_last_dts; ///上一帧的 DTS
char *codec_whitelist; //可用的编解码器的白名单
//省略部分不常用的属性
} AVCodecContext;

```

8.1.5 AVPacket

FFmpeg 用 AVPacket 来存放编码后的视频帧数据, AVPacket 保存了解复用之后、解码之前的数据 (仍然是压缩后的数据) 和关于这些数据的一些附加信息, 如显示时间戳 (PTS)、解码时间戳 (DTS)、数据时长、所在媒体流的索引等。

对于视频 (Video) 来说, AVPacket 通常包含一个压缩的帧, 而音频 (Audio) 则有可能包含多个压缩的帧。并且, 一个 Packet 有可能是空的, 不包含任何压缩数据, 只含有 side data (side data 指的是容器提供的关于 Packet 的一些附加信息。例如, 在编码结束的时候更新一些

流的参数)。

AVPacket 是公共的 ABI (public ABI) 的一部分, 这样的结构体在 FFmpeg 中很少, 由此也可见 AVPacket 的重要性。它可以被分配到栈空间上 (可以使用语句 AVPacket packet; 在栈空间上定义一个 Packet), 并且除非 libavcodec 和 libavformat 有很大的改动, 不然不会在 AVPacket 中添加新的字段。

AVPacket 的位置在 libavcodec/avcodec.h 下, 其代码如下:

```
typedef struct AVPacket {
    AVBufferRef *buf; // AVBufferRef 类型的指针, 用来管理 data 指针引用的数据缓存
    int64_t pts; // 显示时间戳
    int64_t dts; // 解码时间戳
    uint8_t *data; // 指向保存压缩数据的指针, 这就是 AVPacket 实际的数据
    int size;
    int stream_index; // 流索引
    int flags; // 带有 AV_PKT_FLAG 属性的组合
    AVPacketSideData *side_data; // 填充容器的一些附加数据
    int side_data_elems;
    int64_t duration; // Packet 的时长
    int64_t pos; // Packet 中的位置
} AVPacket;
```

AVPacket 实际上可用作一个容器, 它本身并不包含压缩的媒体数据, 而是通过 data 指针引用数据的缓存空间。所以当将一个 Packet 作为参数传递的时候, 就要根据具体的需要, 对 data 引用的这部分数据缓存空间进行特殊的处理。当从一个 Packet 创建另一个 Packet 的时候, 有如下两种情况。

- 两个 Packet 的 data 引用的是同一数据缓存空间, 这时候要注意数据缓存空间的释放问题。
- 两个 Packet 的 data 引用的是不同的数据缓存空间, 每个 Packet 都有数据缓存空间的副本。

在第二种情况下, 数据空间的管理比较简单, 但是数据实际上有多个副本, 这样造成了内存空间的浪费。所以要根据具体的需要, 来选择到底是两个 Packet 共享一个数据缓存空间, 还是每个 Packet 拥有自己独立的缓存空间。

对于多个 Packet 共享同一个缓存空间, FFmpeg 使用了引用计数 (reference-count) 机制。当有新的 Packet 引用共享的缓存空间时, 就将引用计数加 1; 当释放了引用共享空间的 Packet 时, 就将引用计数减 1; 当引用计数为 0 时, 释放引用的缓存空间。

AVPacket 中的 AVBufferRef *buf; 就是用来管理这个引用计数的, AVBufferRef 的声明如下:


```
typedef struct AVBufferRef {
    AVBuffer *buffer;
    uint8_t *data;
    int      size;
} AVBufferRef;
```

在 AVPacket 中使用 AVBufferRef，可以通过两个函数 `av_packet_ref` 和 `av_packet_unref`。

- `av_packet_ref`，函数声明如下：

```
int av_packet_ref(AVPacket *dst, const AVPacket *src)
```

创建一个 `src->data` 的新的引用计数。如果 `src` 已经设置了引用计数 (`src->buffer` 不为空)，直接将其引用计数加 1；如果 `src` 没有设置引用计数 (`src->buffer` 为空)，为 `DST` 创建一个新的引用计数 `buf`，并复制 `src->data` 到 `buf->buffer` 中。最后，复制 `src` 的其他字段到 `DST` 中。

- `av_packet_unref`，函数声明如下：

```
void av_packet_unref(AVPacket *pkt)
```

将缓存空间的引用计数减 1，并将 `Packet` 中的其他字段设为初始值。如果引用计数为 0，则自动释放缓存空间。所以，当两个 `Packet` 共享同一个数据缓存空间的时候，可以这么做。

最后总结一下，`AVPacket` 通常将 `Demuxer` 导出的数据包作为解码器的输入数据，或是收到来自编码器的数据包，通过 `Muxer` 进入输出数据。

图 8-1 表示的是 FFmpeg 复用与解复用的过程。

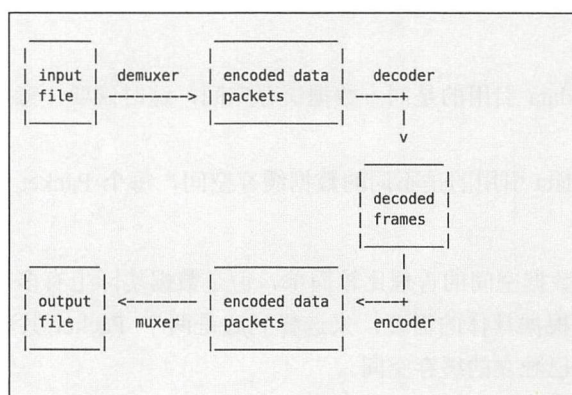


图 8-1 FFmpeg 复用与解复用的过程

8.1.6 AVCodec

AVCodec 是存储编解码器信息的结构体，AVCodec Codec 通过 `avcodec_find_decoder`

(codec_id)找到对应的 Codec, 其在 libavcodec/avcodec.h 位置下:

```
typedef struct AVCodec {
    const char *name; //Codec 的短名称
    const char *long_name; //Codec 的长名称
    enum AVMediaType type; //AVMediaType, 包含音频、视频、字幕等
    enum AVCodecID id; //Codec ID
    int capabilities; //编解码器的播放能力
    const AVRational *supported_framerates; //支持的帧率数组
    const enum AVPixelFormat *pix_fmts; //支持的像素格式数组
    const int *supported_samplerates; //支持的音频采样率
    const enum AVSampleFormat *sample_fmts; //支持的采样格式
    const uint64_t *channel_layouts; //支持的信道布局
    uint8_t max_lowres; //最大的低分辨率
    const AVClass *priv_class; //内部 Context
    const AVProfile *profiles; //能被识别的 profile
    int priv_data_size;
    struct AVCodec *next;
    int (*update_thread_context)(AVCodecContext *dst, const AVCodecContext *src);
    const AVCodecDefault *defaults;
    void (*init_static_data)(struct AVCodec *codec);
    //通过 avcodec_register 初始化 Codec 静态数据
    int (*init)(AVCodecContext *); //初始化
    int (*encode_sub)(AVCodecContext *, uint8_t *buf, int buf_size,
        const struct AVSubtitle *sub);
    int (*encode2)(AVCodecContext *avctx, AVPacket *avpkt, const AVFrame *frame,
        int *got_packet_ptr); //编码数据
    int (*decode)(AVCodecContext *, void *outdata, int *outdata_size, AVPacket
        *avpkt); //解码数据
    int (*close)(AVCodecContext *);
    int (*send_frame)(AVCodecContext *avctx, const AVFrame *frame);
    //发送解码后音视频数据
    int (*send_packet)(AVCodecContext *avctx, const AVPacket *avpkt);
    //发送音视频压缩数据
    int (*receive_frame)(AVCodecContext *avctx, AVFrame *frame);
    //接收解码后音视频数据
    int (*receive_packet)(AVCodecContext *avctx, AVPacket *avpkt);
    //接收音视频压缩数据
    void (*flush)(AVCodecContext *); //刷新缓冲区
    int caps_internal; //内部 Codec 的能力
} AVCodec;
```

8.1.7 AVFrame

AVFrame 结构体一般用于存储原始数据 (即非压缩数据, 例如对视频来说是 YUV、

RGB, 对音频来说是 PCM), 此外还包含了一些相关信息。比如说, 解码的时候存储了宏块类型表、QP 表、运动矢量表等数据。编码的时候也存储了相关数据。AVFrame 结构体代码如下:

```
typedef struct AVFrame {
    uint8_t *data[AV_NUM_DATA_POINTERS];
    int linesize[AV_NUM_DATA_POINTERS];
    uint8_t **extended_data; // 扩展的数据
    int width, height; // 视频帧的宽和高
    int nb_samples; // 每个信道的音频采样点个数
    int format; // 帧的像素格式
    int key_frame; // 1 表示关键帧, 0 表示非关键帧
    enum AVPictureType pict_type; // 帧的图像类型
    AVRational sample_aspect_ratio; // 视频帧的采样率
    int64_t pts; // 显示时间戳
    int64_t pkt_pts; // 从 AVPacket 中复制的 PTS
    int64_t pkt_dts; // 从 AVPacket 中复制的 DTS
    int coded_picture_number; // 按解码排序后的图像数
    int display_picture_number; // 按显示位置排序的图像数
    int quality; // 在 1~FF_LAMBDA_MAX(256*128-1) 之间取值
    void *opaque; // 私有数据
    int repeat_pict;
    // 解码时有多少个图像被延迟, extra_delay = repeat_pict / (2*fps)
    int interlaced_frame; // 交错帧, 表示图像内容是交错的
    int sample_rate; // 音频数据的采样率
    uint64_t channel_layout; // 音频信道布局
    enum AVColorSpace colorspace; // YUV 颜色空间类型
    int64_t pkt_pos; // 记录上一个包输出解码器时 Packet 的位置
    int64_t pkt_duration; // Packet 的时长
    int channels; // 音频信道个数
    int pkt_size;
    // Packet 的大小, 由 av_frame_get_pkt_size 获取、av_frame_set_pkt_size 设置
} AVFrame;
```

AVFrame 存放从 AVPacket 中解码出来的原始数据, 其必须通过 `av_frame_alloc` 来创建, 通过 `av_frame_free` 来释放。和 AVPacket 类似, AVFrame 中也有一块数据缓存空间, 在调用 `av_frame_alloc` 的时候并不会为这块缓存区域分配空间, 需要使用其他的方法。在解码的过程中使用了两个 AVFrame, 这两个 AVFrame 分配缓存空间的方法并不相同, 下面分别进行介绍。

- 一个 AVFrame 用来存放从 AVPacket 中解码出来的原始数据, 这个 AVFrame 的数据缓存空间通过调用 `avcodec_decode_video` 来分配和填充。

- 另一个 AVFrame 用来存放将解码出来的原始数据变换为需要的数据格式（例如 RGB、RGBA）的数据，这个 AVFrame 需要手动分配数据缓存空间，代码如下：

```
AVFrame* pFrameYUV;
pFrameYUV = av_frame_alloc();
//手动为 pFrameYUV 分配数据缓存空间
int numBytes = avpicture_get_size(AV_PIX_FMT_YUV420P, pCodecCtx->width,
pCodecCtx->height);
uint8_t* buffer = (uint8_t*)av_malloc(numBytes * sizeof(uint8_t));
//将分配的数据缓存空间和 AVFrame 关联起来
avpicture_fill((AVPicture *)pFrameYUV, buffer, AV_PIX_FMT_YUV420P, pCodecCtx
->width, pCodecCtx->height)
```

首先计算需要的缓存空间大小，调用 `av_malloc` 分配缓存空间，最后调用 `avpicture_fill` 将分配的缓存空间和 AVFrame 关联起来。

调用 `av_frame_free` 来释放 AVFrame，该函数不只会释放 AVFrame 本身的空间，还会释放包含在其内的其他对象动态申请的空间，例如上面的缓存空间。

`av_malloc` 和 `av_free`，FFmpeg 并没有提供垃圾回收机制，所有的内存管理都要手动进行。只是 `av_malloc` 在申请内存空间的时候会考虑到内存对齐（2 字节、4 字节对齐），其申请的空间要调用 `av_free` 释放。

最后总结一下。

- 这个结构体用来描述解码出的音视频数据。
- AVFrame 必须使用 `av_frame_alloc` 分配。
- AVFrame 必须使用 `av_frame_free` 释放。
- AVFrame 通常分配一次，然后重复使用多次，当不同的帧数据（注：如一个 AVFrame 持有来自解码后的帧）再次被使用时，`av_frame_unref` 将自由持有任何之前的帧引用并重置它变成初始态。
- 一个 AVFrame 所描述的数据通常是通过参考 AVBuffer API 计算的。内部的 Buffer 引用存储在 AVFrame.buf/AVFrame.extended_buf 中。
- AVFrame 将用于引用计数，当至少一个引用被设置时，如果 AVFrame.buf[0] != NULL，每个数据至少包含一个 AVFrame.buf/AVFrame.extended_buf。
- sizeof(AVFrame) 不是一个公有 API，因此新的成员将被添加到末尾。同样字段标记为只访问 `av_opt_ptr` 就可以重新排序了。
- `uint8_t *data[AV_NUM_DATA_POINTERS]`：指针数组，存放 YUV 数据的地方。如图 8-2 所示，一般占用前 3 个指针，分别指向 Y、U、V 数据。

	名称	值	类型
0x	pframe	0x02fafe00 (data=0x02fafe00 filesize=0x02fafe20 extended_data=0x02fafe00...)	AVFrame*
0x	data	0x02fafe00	unsigned char*[8]
0x	[0]	0x00000000 '\0\0\0\0\0\0\0\0'	unsigned char*
0x	[1]	0x0309f6ff "百度网景公司中国网络中心中国网络中心中国网络中心中国网络中心"	unsigned char*
0x	[2]	0x0301e6ff "....."	unsigned char*
0x	[3]	0x00000000 <错误的指针>	unsigned char*
0x	[4]	0x00000000 <错误的指针>	unsigned char*
0x	[5]	0x00000000 <错误的指针>	unsigned char*
0x	[6]	0x00000000 <错误的指针>	unsigned char*
0x	[7]	0x00000000 <错误的指针>	unsigned char*
0x	avcodec	0x02fafe20	int[8]
0x	[0]	784	int
0x	[1]	352	int
0x	[2]	352	int
0x	[3]	0	int
0x	[4]	0	int
0x	[5]	0	int
0x	[6]	0	int
0x	[7]	0	int
0x	extended_data	0x02fafe00	unsigned char**
0x	widht	672	int
0x	height	272	int
0x	nb_samples	0	int
0x	format	0	int
0x	key_frame	1	int
0x	pic_type	AV_PICTURE_TYPE_I	AVPictureType
0x	samples_per_picture	{num=1,den=1}	AVRational
0x	pts	-9223372036854775808	_int64
0x	pkt_pts	14400	_int64
0x	pkt_dts	14400	_int64
0x	coded_picture_number	0	int
0x	display_picture_number	0	int
0x	quality	0	int
0x	opaque	0x00000000	void*
0x	error	0x02fafe90	unsigned_int64[8]
0x	repeat_pict	0	int

图 8-2 某一帧 AVFrame 结构体中的变量赋值图

- 对于 packed 格式的数据（例如 RGB24），会存放到 data[0]里面。
- 对于 planar 格式的数据（例如 YUV420P），会分开成 data[0]、data[1]、data[2]……（YUV420P 中 data[0]存储 Y，data[1]存储 U，data[2]存储 V）。
- int linesize[AV_NUM_DATA_POINTERS]：图像各个分量数据在此结构体中的宽度。注意，这并不是图像的宽度。在此例中图像的尺寸为 672×272，而亮度分量的宽度为 704，应该是图像宽度经过 64 位对齐后的结果。

8.1.8 AVIOContext

协议（文件）操作的顶层结构是 AVIOContext，这个对象实现了带缓冲的读/写操作。FFmpeg 的输入对象 AVFormat 的 pb 字段指向一个 AVIOContext。

AVIOContext 的 opaque 实际指向一个 URLContext 对象，代码如下：

```
typedef struct AVIOContext {
const AVClass *av_class;//AVIOContext 通过 avio_open2 函数创建, av_class 通
//过 option 选项传递到 protocols (协议)
unsigned char *buffer; //起始 Buffer
int buffer_size;//最大的 Buffer 大小
unsigned char *buf_ptr;//当前 Buffer 的位置
unsigned char *buf_end;//Buffer 的最后位置
int (*read_packet)(void *opaque, uint8_t *buf, int buf_size);
//读取音视频压缩数据
int (*write_packet)(void *opaque, uint8_t *buf, int buf_size);
//写入音视频压缩数据
int64_t (*seek)(void *opaque, int64_t offset, int whence);//快进/快退
```

```

int64_t pos; //当前 Buffer 中的文件位置
int must_flush; //下一个快进/快退是否需要刷新
int eof_reached; //确定是否 EOF (End Of File)
int write_flag; //确定是否写
int max_packet_size;
unsigned long checksum;
unsigned char *checksum_ptr;
unsigned long (*update_checksum)(unsigned long checksum, const uint8_t *buf,
unsigned int size);
int error; //包含一些 errorcode
int (*read_pause)(void *opaque, int pause); //暂停
int64_t (*read_seek)(void *opaque, int stream_index, int64_t timestamp, int
flags); //快进/快退到给定的时间点
int seekable; //当前流是否能快进/快退
int64_t maxsize; //最大文件大小, 用于限制分配空间
int64_t bytes_read; //读取 bytes 次数统计
int seek_count; //快进/快退次数统计
int writeout_count; //写入数据次数统计
int orig_buffer_size; //原始 Buffer 大小
const char *protocol_whitelist; //支持的协议列表
const char *protocol_blacklist; //不支持的协议列表
int (*write_data_type)(void *opaque, uint8_t *buf, int buf_size,
enum AVIODataMarkerType type, int64_t time);
//一个回调函数, 用于代替 write_packet
int64_t last_time; //上次的时间点
} AVIOContext;

```

一般情况下, FFmpeg 支持打开一个本地文件, 例如 “C:\xxx.avi”, 或者一个流媒体协议的 URL, 例如 “rtmp://222.197.224.44/live/”。其打开文件的函数是 `avformat_open_input`, 直接将文件路径或者流媒体 URL 的字符串传递给该函数就可以了。

典型应用如下:

```

AVFormatContext *ic = NULL;
ic = avformat_alloc_context();
unsigned char *iobuffer = (unsigned char *)av_malloc(32768);
AVIOContext *avio = avio_alloc_context(iobuffer, 32768, 0, buffer, fill_
iobuffer, NULL, NULL);
ic->pb = avio;
err = avformat_open_input(&ic, is->filename, is->iformat, &format_opts);

```

8.1.9 URLProtocol

URLProtocol 是 FFmpeg 操作文件的结构 (包括文件、网络数据流等), 包括 `open`、`close`、`read`、`write`、`seek` 等操作。URLProtocol 为协议操作对象, 针对每种协议会有一个这样



的对象，每个协议操作对象和一个协议对象关联。

在 `av_register_all` 函数中，通过调用 `REGISTER_PROTOCOL` 宏，所有的 `URLProtocol` 都保存在以 `first_protocol` 为链表头的链表中。`URLProtocol` 的结构体定义如下：

```
typedef struct URLProtocol {
    const char *name; // 协议名
    int (*url_open)(URLContext *h, const char *url, int flags);
    int (*url_open2)(URLContext *h, const char *url, int flags, AVDictionary
**options); // 打开网络协议的回调函数
    int (*url_accept)(URLContext *s, URLContext **c);
    int (*url_handshake)(URLContext *c);
    int (*url_read)(URLContext *h, unsigned char *buf, int size);
    // 读取协议中的数据，如果是 EOF 或发生错误，返回 AVERROR
    int (*url_write)(URLContext *h, const unsigned char *buf, int size);
    int64_t (*url_seek)(URLContext *h, int64_t pos, int whence);
    // 快进/快退到某个点
    int (*url_close)(URLContext *h); // 关闭
    int (*url_read_pause)(URLContext *h, int pause); // 暂停读取数据
    int64_t (*url_read_seek)(URLContext *h, int stream_index,
int64_t timestamp, int flags); // seek 操作
    int (*url_get_file_handle)(URLContext *h);
    int (*url_get_multi_file_handle)(URLContext *h, int **handles,
int *numhandles);
    int (*url_shutdown)(URLContext *h, int flags);
    int priv_data_size;
    const AVClass *priv_data_class;
    int flags;
    int (*url_check)(URLContext *h, int mask);
    int (*url_open_dir)(URLContext *h); // 打开某个路径地址
    int (*url_read_dir)(URLContext *h, AVIODirEntry **next);
    // 读取某个路径文件数据
    int (*url_close_dir)(URLContext *h);
    int (*url_delete)(URLContext *h);
    int (*url_move)(URLContext *h_src, URLContext *h_dst);
    const char *default_whitelist;
} URLProtocol;
```

8.1.10 URLContext

`URLContext` 对象封装了协议对象及协议操作对象，`URLContext` 在 `avio.c` 中通过 `url_alloc_for_protocol` 进行初始化，并且分配空间（使用 `av_malloc(sizeof(URLContext)+strlen(filename)+1)` 函数）。`URLContext` 结构体代码如下：



```
typedef struct URLContext {
    const AVClass *av_class; //通过 url_open 函数设置进入, 主要用于 av_log
    const struct URLProtocol *prot; //指向具体的协议操作对象
    void *priv_data; //指向具体的协议对象
    char *filename; //具体的 URL
    int flags;
    int max_packet_size; //最大的 Packet (音视频压缩数据) 大小
    int is_streamed; //默认是 false, 若该值是 true, 表示的是 Stream
    int is_connected;
    AVIOInterruptCB interrupt_callback;
    int64_t rw_timeout; //最大网络等待读/写时间
    const char *protocol_whitelist; //可支持的协议列表
    const char *protocol_blacklist; //不支持的协议列表
} URLContext;
```

8.2 FFmpeg 关键函数介绍

定义识别文件格式和媒体类型库使用的宏、数据结构和函数, 通常这些宏、数据结构和函数在 ffmpeg 模块内相对全局有效, 因为用结构体定义的类都属于 public 类型。

8.2.1 av_register_all 函数

libavformat 是用来处理多种媒体格式的, 主要有两个作用, 一个是分离音视频, 一个是反向合成具体的媒体格式。同时其也支持 I/O 模块, 支持一系列的协议获取数据 (文件、TCP、HTTP 等), 在使用 libavformat 之前, 需要先调用 av_register_all 函数来注册所有编译的 muxers、demuxers 及 protocols, 如果要使用 libavformat 的网络接口, 需要使用 avformat_network_init 接口。ffplay.c 的 av_register_all 函数调用后的时序图如图 8-3 所示。

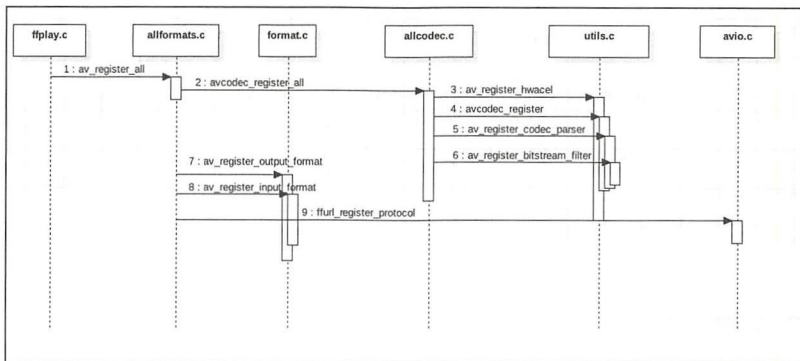


图 8-3 av_register_all 函数调用后的时序图



其中, allcodec.c 及 utils.c 在 libcodec 目录下。

8.2.2 avformat_alloc_context 函数

当使用 AVFormatContext 时, 需要对 AVFormatContext 分配内存空间, avformat_alloc_context 函数的作用就基于此, 其代码如下:

```
AVFormatContext *avformat_alloc_context(void)
{
    AVFormatContext *ic;
    ic = av_malloc(sizeof(AVFormatContext)); //为 AVFormatContext 分配空间
    if (!ic) return ic;
    avformat_get_context_defaults(ic);
    ic->internal = av_mallocz(sizeof(*ic->internal));
    //为 AVFormatContext 内部 Context 分配空间
    if (!ic->internal) {
        avformat_free_context(ic);
        return NULL;
    }
    ic->internal->offset = AV_NOPTS_VALUE; //赋初值
    ic->internal->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;
    //大小为 2 500 000
    return ic;
}
```

从代码中可以看出, avformat_alloc_context 调用 av_malloc 为 AVFormatContext 结构体分配了内存, 而且同时也给 AVFormatContext 中的 internal 字段分配了内存 (这个字段是 FFmpeg 内部使用的, 先不分析)。此外调用了 avformat_get_context_defaults 函数。该函数用于设置 AVFormatContext 的字段的默认值。它的定义也位于 libavformat/options.c, 确切的说, 就位于 avformat_alloc_context 上面。我们看一下该函数的定义:

```
static void avformat_get_context_defaults(AVFormatContext *s)
{
    memset(s, 0, sizeof(AVFormatContext)); //内存空间初始化
    s->av_class = &av_format_context_class;
    s->io_open = io_open_default;
    s->io_close = io_close_default;
    av_opt_set_defaults(s);
}
```

8.2.3 avio_open 函数

该函数用于打开 FFmpeg 的输入/输出文件, 其在 libavformat/aviobuf.c 文件中, 代码如下:



```
int avio_open(AVIOContext **s, const char *filename, int flags)
{
    return avio_open2(s, filename, flags, NULL, NULL);
}
```

其中主要调用了 `avio_open2` 函数，该函数最后两个参数传入的都是 `NULL`，代码如下：

```
int avio_open2(AVIOContext **s, const char *filename, int flags,
               const AVIOInterruptCB *int_cb, AVDictionary **options)
{
    return ffio_open_whitelist(s, filename, flags, int_cb, options, NULL,
NULL);
}
```

`avio_open2` 函数参数的含义如下。

- `s`: 函数调用成功之后创建的 `AVIOContext` 结构体。
- `filename`: 文件名。
- `flags`: 打开地址的方式，可以选择只读、只写或者读/写，取值如下。
 - `AVIO_FLAG_READ`: 只读。
 - `AVIO_FLAG_WRITE`: 只写。
 - `AVIO_FLAG_READ_WRITE`: 读/写。
- `int_cb`: 没有使用，传入 `NULL`。
- `options`: 没有使用，传入 `NULL`。

从 `avio_open2` 函数的源代码可以看出，它主要调用了 `ffio_open_whitelist`，传入了文件名及一些标志。其中主要调用了 `ffurl_open` 和 `ffio_fdopen` 函数，`ffurl_open` 用于初始化 `URLContext`。`ffio_open_whitelist` 函数的代码如下：

```
int ffio_open_whitelist(AVIOContext **s, const char *filename, int flags,
                       const AVIOInterruptCB *int_cb, AVDictionary **options, const char
                       *whitelist, const char *blacklist) {
    URLContext *h;
    int err;
    err = ffurl_open_whitelist(&h, filename, flags, int_cb, options, white-
list, blacklist, NULL);
    if (err < 0)
        return err;
    err = ffio_fdopen(s, h);
    if (err < 0) {
        ffurl_close(h);
        return err;
    }
    return 0;
}
```



```
}
```

可以看出, `ffio_open_whitelist` 中主要调用了两个函数, 一个是 `ffurl_open_whitelist`, 另外一个则是 `ffio_fdopen`, `ffio_fdopen` 用于根据 `URLContext` 初始化 `AVIOContext`。 `URLContext` 中包含的 `URLProtocol` 完成了具体的协议读/写等工作。 `AVIOContext` 则是在 `URLContext` 的读/写函数外面封装了一层。 `ffio_fdopen` 函数的代码如下:

```
int ffio_fdopen(AVIOContext **s, URLContext *h)
{
    AVIOInternal *internal = NULL;
    uint8_t *buffer = NULL;
    int buffer_size, max_packet_size;
    max_packet_size = h->max_packet_size;
    if (max_packet_size) {
        buffer_size = max_packet_size; // 不需要缓冲多个数据包
    } else {
        buffer_size = IO_BUFFER_SIZE; // 定义大小为 32 768
    }
    buffer = av_malloc(buffer_size); // 给 Buffer 分配 buffer_size 个大小
    if (!buffer)
        return AVERERROR(ENOMEM);
    internal = av_mallocz(sizeof(*internal));
    if (!internal)
        goto fail;
    internal->h = h;
    *s = avio_alloc_context(buffer, buffer_size, h->flags & AVIO_FLAG_
WRITE, internal, io_read_packet, io_write_packet, io_seek);
    if (!*s)
        goto fail;
    (*s)->protocol_whitelist = av_strdup(h->protocol_whitelist);
    if (!(*s)->protocol_whitelist && h->protocol_whitelist) {
        avio_closep(s);
        goto fail;
    }
    (*s)->protocol_blacklist = av_strdup(h->protocol_blacklist);
    if (!(*s)->protocol_blacklist && h->protocol_blacklist) {
        avio_closep(s);
        goto fail;
    }
    (*s)->direct = h->flags & AVIO_FLAG_DIRECT;
    (*s)->seekable = h->is_streamed ? 0 : AVIO_SEEKABLE_NORMAL;
    // 是否能快进/快退赋值
    (*s)->max_packet_size = max_packet_size;
    if (h->prot) {
```




```
        (*s)->read_pause = io_read_pause;//暂停
        (*s)->read_seek = io_read_seek;//快进/快退
    }
    (*s)->av_class = &ff_avio_class;
    return 0;
fail:
    av_freep(&internal);//释放相关资源
    av_freep(&buffer);
    return AVERERROR(ENOMEM);
}
```

代码中的部分是对不支持协议的判断。另外一部分就是赋初始值，分配一些结构体空间，其中的 `avio_alloc_context` 就用于对 `AVIOContext` 分配内存空间。

8.2.4 avformat_open_input 函数

`avformat_open_input` 的主要功能是打开一个文件，读取 `header`，不会涉及打开解码器。与之对应的是通过 `avformat_close_input` 函数关闭文件。`avformat_open_input` 函数的代码如下：

```
int avformat_open_input(AVFormatContext **ps, const char *filename,
                        AVInputFormat *fmt, AVDictionary **options)
{
    AVFormatContext *s = *ps;
    int i, ret = 0;
    AVDictionary *tmp = NULL;
    ID3v2ExtraMeta *id3v2_extra_meta = NULL;
    //省略部分代码
    if ((ret = av_opt_set_dict(s, &tmp)) < 0)
        goto fail;
    if ((ret = init_input(s, filename, &tmp)) < 0)
        //调用 init_input, 绝大部分初始化操作在这里做
        goto fail;
    s->probe_score = ret;
    //将 ret (即 init_input 是否执行成功) 赋值到 AVFormatContext 的 probe_score
    //成员变量
    if (!s->protocol_whitelist && s->pb && s->pb->protocol_whitelist) {
        s->protocol_whitelist = av_strdup(s->pb->protocol_whitelist);
        if (!s->protocol_whitelist) {
            ret = AVERERROR(ENOMEM);
            goto fail;
        }
    }
    if (!s->protocol_blacklist && s->pb && s->pb->protocol_blacklist) {
        s->protocol_blacklist = av_strdup(s->pb->protocol_blacklist);
```




```
        if (!s->protocol_blacklist) {
            ret = AVERERROR(ENOMEM);
            goto fail;
        }
    }
    if (s->format_whitelist && av_match_list(s->iformat->name, s-> format_
whitelist, ',') <= 0) {
        av_log(s, AV_LOG_ERROR, "Format not on whitelist \'%s\'\\n", s->
format_whitelist);
        ret = AVERERROR(EINVAL);
        goto fail;
    }
    avio_skip(s->pb, s->skip_initial_bytes); //跳过 skip_initial_bytes
    //检查文件名, 过滤掉图像格式
    if (s->iformat->flags & AVFMT_NEEDNUMBER) {
        if (!av_filename_number_test(filename)) {
            ret = AVERERROR(EINVAL);
            goto fail;
        }
    }
    //给 duration 和 start_time 赋 0 值
    s->duration = s->start_time = AV_NOPTS_VALUE;
    //保存文件名
    av_strlcpy(s->filename, filename ? filename : "", sizeof(s-> filename));
    //分配私有数据, 用于读/写格式中的私有数据
    if (s->iformat->priv_data_size > 0) {
        if (!(s->priv_data = av_mallocz(s->iformat->priv_data_size))) {
            ret = AVERERROR(ENOMEM);
            goto fail;
        }
        if (s->iformat->priv_class) {
            *(const AVClass **) s->priv_data = s->iformat->priv_class;
            av_opt_set_defaults(s->priv_data);
            if ((ret = av_opt_set_dict(s->priv_data, &tmp)) < 0)
                goto fail;
        }
    }
}
/* 举例, . AVFMT_NOFILE 格式不会有 AVIOContext */
if (s->pb)
    ff_id3v2_read(s, ID3v2_DEFAULT_MAGIC, &id3v2_extra_meta, 0);
if (!(s->flags&AVFMT_FLAG_PRIV_OPT) && s->iformat->read_header)
    if ((ret = s->iformat->read_header(s)) < 0)
        //读取多媒体数据文件头, 根据音视频创建相应的 AVStream
        goto fail;
```



```
if (id3v2_extra_meta) { //特殊格式 extra_meta 的处理
    if (!strcmp(s->iformat->name, "mp3") || !strcmp(s->iformat->name,
"aac") || !strcmp(s->iformat->name, "tta")) {
        if ((ret = ff_id3v2_parse_apic(s, &id3v2_extra_meta)) < 0)
            goto fail;
    } else
        av_log(s, AV_LOG_DEBUG, "demuxer does not support additional
id3 data, skipping\n");
}
ff_id3v2_free_extra_meta(&id3v2_extra_meta);
if ((ret = avformat_queue_attached_pictures(s)) < 0)
    goto fail;
if (!(s->flags&AVFMT_FLAG_PRIV_OPT) && s->pb && !s->internal-> data_
offset)
    s->internal->data_offset = avio_tell(s->pb); //保存数据区起始位置
s->internal->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;
update_stream_avctx(s); //更新 Stream 信息
for (i = 0; i < s->nb_streams; i++)
    s-> streams[i] -> internal -> orig_codec_id = s->streams[i] ->
codecpar -> codec_id;
//将 codec_id 赋给 AVStream 内部的 Context 所指向的 orig_codec_id
if (options) {
    av_dict_free(options);
    *options = tmp;
}
*ps = s;
return 0;
fail:
ff_id3v2_free_extra_meta(&id3v2_extra_meta);
av_dict_free(&tmp);
if (s->pb && !(s->flags & AVFMT_FLAG_CUSTOM_IO))
    avio_closep(&s->pb);
avformat_free_context(s);
*ps = NULL;
return ret;
}
```

其中两个内部重点函数就是 `init_input` 和 `s->iformat->read_header` 函数。对于 `init_input` 函数来说, 除上面代码注释中的要点, 还涉及读/写文件相关的结构体, 如 `AVIOContext` 及 `AVInputFormat` 等。另外一个 `s->iformat->read_header` 函数主要做某种格式初始化工作, 比如填充自己的私有结构, 根据 `Stream` 的数量分配 `Stream` 结构并进行初始化, 把文件指针指向数据区开始初始化。



8.2.5 avformat_find_stream_info 函数

由于该函数很长，我们对它进行拆解分析，首先看看定义的变量，代码如下：

```
int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options)
{
    int i, count = 0, ret = 0, j;
    int64_t read_size;
    AVStream *st;
    AVCodecContext *avctx;
    AVPacket pkt1, *pkt;
    int64_t old_offset = avio_tell(ic->pb);

    int orig_nb_streams = ic->nb_streams;
    int flush_codecs;
    int64_t max_analyze_duration = ic->max_analyze_duration;
    int64_t max_stream_analyze_duration;
    int64_t max_subtitle_analyze_duration;
    int64_t probesize = ic->probesize; // 探测播放器能够播放的数据大小
    int eof_reached = 0;
    flush_codecs = probesize > 0;
```

上面的代码里定义了一些参数，分别介绍如下。

`probesize` 用于限制 `avformat_find_stream_info` 接口内部读取的最大数据量，即播放器起播的数据，为了获取流的信息，这个函数会尝试读一些包（Packet），然后分析这些数据是否能达到起播条件。

`orig_nb_streams` 表示这个文件中流的数量。一部影片可能包含 3 个流，分别为音频流、视频流和字幕流。

下面的代码主要是流数据最大时长分析过程：

```
av_opt_set(ic, "skip_clear", "1", AV_OPT_SEARCH_CHILDREN);
max_stream_analyze_duration = max_analyze_duration;
max_subtitle_analyze_duration = max_analyze_duration;
if (!max_analyze_duration) {
    max_stream_analyze_duration =
        max_analyze_duration = 5*AV_TIME_BASE;
    max_subtitle_analyze_duration = 30*AV_TIME_BASE;
    if (!strcmp(ic->iformat->name, "flv"))
        max_stream_analyze_duration = 90*AV_TIME_BASE;
    if (!strcmp(ic->iformat->name, "mpeg") || !strcmp(ic->iformat->
name, "mpegts"))
        max_stream_analyze_duration = 7*AV_TIME_BASE;
```




```
}
```

以上代码中定义了最大分析时长的限制。max_stream_analyze_duration 等于 max_analyze_duration, 也等于 timebase 的 5 倍。此外, 如果文件格式为 FLV 或者 MPEG, 那么它们都会有各自的最大分析时长的限制。调用 avio_tell 函数读取文件大小, 代码如下:

```
if (ic->pb)
    av_log(ic, AV_LOG_DEBUG, "Before avformat_find_stream_info() pos:
    %"PRIu64" bytes read:%"PRIu64" seeks:%d nb_streams:%d\n", avio_tell(ic-> pb),
    ic->pb->bytes_read, ic->pb->seek_count, ic->nb_streams);
```

pb 是 AVIOContext, avio_tell 里面调用是 avio_seek, 实际上类似于 C++ 中的 fseek 函数, 获取文件内容信息, 获取到信息后开始遍历文件中的几条流数据, 代码如下:

```
for (i = 0; i < ic->nb_streams; i++) {
    const AVCodec *codec;
    AVDictionary *thread_opt = NULL;
    st = ic->streams[i];
    avctx = st->internal->avctx; //给内部 AVContext 赋值
    if (st->codecpar->codec_type == AVMEDIA_TYPE_VIDEO ||
        st->codecpar->codec_type == AVMEDIA_TYPE_SUBTITLE) {
        /*if (!st->time_base.num)
            st->time_base = */
        if (!avctx->time_base.num)
            avctx->time_base = st->time_base;
    }

    if (!st->parser && !(ic->flags & AVFMT_FLAG_NOPARSE) && st->
request_probe <= 0) {
        st->parser = av_parser_init(st->codecpar->codec_id);
        if (st->parser) {
            if (st->need_parsing == AVSTREAM_PARSE_HEADERS) {
                st->parser->flags |= PARSER_FLAG_COMPLETE_FRAMES;
            } else if (st->need_parsing == AVSTREAM_PARSE_FULL_RAW) {
                st->parser->flags |= PARSER_FLAG_USE_CODEC_TS;
            }
        } else if (st->need_parsing) {
            av_log(ic, AV_LOG_VERBOSE, "parser not found for codec "
                "%s, packets or times may be invalid.\n",
                avcodec_get_name(st->codecpar->codec_id));
        }
    }
}

/*检查调用方是否覆盖了编解码器 ID*/
#if FF_API_LAVF_AVCTX
FF_DISABLE_DEPRECATED_WARNINGS
```

```

        if (st->codec->codec_id != st->internal->orig_codec_id) {
            st->codecpa->codec_id = st->codec->codec_id;
            st->codecpa->codec_type = st->codec->codec_type;
            st->internal->orig_codec_id = st->codec->codec_id;
        }
FF_ENABLE_DEPRECATION_WARNINGS
#endif

        if (st->codecpa->codec_id != st->internal->orig_codec_id)
            st->internal->orig_codec_id = st->codecpa->codec_id;
        ret = avcodec_parameters_to_context(avctx, st->codecpa);
        if (ret < 0)
            goto find_stream_info_err;
        if (st->request_probe <= 0)
            st->internal->avctx_inited = 1;
        codec = find_decoder(ic, st, st->codecpa->codec_id);
        /*强制使线程计数为 1, 因为 H.264 解码器不会提取 SPS 和 PPS 到多线程解码*/
        av_dict_set(options ? &options[i] : &thread_opt, "threads", "1", 0);
        if (ic->codec_whitelist)
            av_dict_set(options ? &options[i] : &thread_opt, "codec_
whitelist", ic->codec_whitelist, 0);
        /*确保 subtitle_header 设置正确*/
        if (st->codecpa->codec_type == AVMEDIA_TYPE_SUBTITLE
            && codec && !avctx->codec) {
            if (avcodec_open2(avctx, codec, options ? &options[i] : &thread_
opt) < 0)
                av_log(ic, AV_LOG_WARNING,
                    "Failed to open codec in av_find_stream_info\n");
        }

        //试着打开解码器, 以便得到足够多的参数
        if (!has_codec_parameters(st, NULL) && st->request_probe <= 0) {
            if (codec && !avctx->codec)
                if (avcodec_open2(avctx, codec, options ? &options[i] :
&thread_opt) < 0)
                    av_log(ic, AV_LOG_WARNING, "Failed to open codec in
av_find_stream_info\n");
        }
        if (!options)
            av_dict_free(&thread_opt);
    }
}

```

以上代码进行了如下这些操作。

(1) 获得编码器上下文环境 avctx, 设置 avctx 的 time_base、code_id、codec_type、orig_codec_id。



(2) 如果解析器 `paser` 为空, 那么会初始化解码器。

(3) 把解析器中的参数赋值到编解码器上下文环境中, 调用的是 `avcodec_parameters_to_context` 函数。

(4) 通过 `find_decoder` 函数, 根据 `codec_id` 查找 Codec。`find_decoder` 的功能就是遍历一个编解码器的链表, 其首个结构是 `first_avcodec`。编解码器链表是我们在 `av_register_all` 函数中注册的。`find_decoder` 匹配每一个编解码器的 ID, 找到后返回给对应的编解码器。

(5) 通过 `avcodec_open2` 函数传入找到的 Codec。

总结一下就是, 通过初始化 `time_base`、`codec_id` 等参数, 初始化解码器 `paser`, 然后针对每一个流, 根据 `codec_id` 找到对应的编解码器并打开。也就是说, 第一次遍历数据流后, 使得我们对于每一个数据流 (视频流、音频流、字幕流) 都有一个 Codec 可用。接下来继续遍历:

```
for (i = 0; i < ic->nb_streams; i++) {
    #if FF_API_R_FRAME_RATE
        ic->streams[i]->info->last_dts = AV_NOPTS_VALUE;
    #endif
        ic->streams[i]->info->fps_first_dts = AV_NOPTS_VALUE;
        ic->streams[i]->info->fps_last_dts = AV_NOPTS_VALUE;
}
```

第二次遍历流是在 Codec 已经打开之后, 对于每一个流设置了一些参数。给 `info->last_dts` 赋初始值, 也给 `fps_first_dts` 及 `fps_last_dts` 赋初始值:

```
read_size = 0;
for (;;) {
    int analyzed_all_streams;
    if (ff_check_interrupt(&ic->interrupt_callback)) {
        ret = AERROR_EXIT;
        av_log(ic, AV_LOG_DEBUG, "interrupted\n");
        break;
    }
    /*检查是否还有一个编解码器需要处理*/
    for (i = 0; i < ic->nb_streams; i++) {
        int fps_analyze_framecount = 20;
        st = ic->streams[i];
        if (!has_codec_parameters(st, NULL))
            break;
        /*如果时间基是不精确的 (如 MKV 格式通常是毫秒级精度的), 那么需要分析更多的帧, 以可靠地到达正确的帧率*/
        if (av_q2d(st->time_base) > 0.0005)
            fps_analyze_framecount *= 2;
```




```
if (!tb_unreliable(st->internal->avctx))
    fps_analyze_framecount = 0;
if (ic->fps_probe_size >= 0)
    fps_analyze_framecount = ic->fps_probe_size;
if (st->disposition & AV_DISPOSITION_ATTACHED_PIC)
    fps_analyze_framecount = 0;
/* variable fps and no guess at the real fps */
if (!(st->r_frame_rate.num && st->avg_frame_rate.num) &&
    st->codecpars->codec_type == AVMEDIA_TYPE_VIDEO) {
    int count = (ic->iformat->flags & AVFMT_NOTIMESTAMPS) ?
        st->info->codec_info_duration_fields/2 :
        st->info->duration_count;
    if (count < fps_analyze_framecount)
        break;
}
if (st->parser && st->parser->parser->split &&
    !st->codecpars->extradata)
    break;
if (st->first_dts == AV_NOPTS_VALUE &&
    !(ic->iformat->flags & AVFMT_NOTIMESTAMPS) &&
    st->codec_info_nb_frames < ((st->disposition & AV_DISPOSITION_
    ATTACHED_PIC) ? 1 : ic->max_ts_probe) &&
    (st->codecpars->codec_type == AVMEDIA_TYPE_VIDEO ||
    st->codecpars->codec_type == AVMEDIA_TYPE_AUDIO))
    break;
}
analyzed_all_streams = 0;
if (i == ic->nb_streams) {
    analyzed_all_streams = 1;
    /*注意，如果封装格式没有头信息，那么我们需要读取一些 Packet 来获取更多数据流*/
    if (!(ic->ctx_flags & AVFMTCTX_NOHEADER)) {
        //如果找到所有编解码器的信息，就直接跳出
        ret = count;
        av_log(ic, AV_LOG_DEBUG, "All info found\n");
        flush_codecs = 0;
        break;
    }
}
/*我们没有得到所有的编解码器信息，但是已经读取了很多数据*/
if (read_size >= probesize) {
    ret = count;
    av_log(ic, AV_LOG_DEBUG,
        "Probe buffer size limit of %"PRIu64" bytes reached\n",
        probesize);
}
```



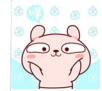
```
    for (i = 0; i < ic->nb_streams; i++)
        if (!ic->streams[i]->r_frame_rate.num &&
            ic->streams[i]->info->duration_count <= 1 &&
            ic->streams[i]->codecpars->codec_type == AVMEDIA_TYPE_
VIDEO && strcmp(ic->iformat->name, "image2"))
            av_log(ic, AV_LOG_WARNING, "Stream #d: not enough frames
to estimate rate; " "consider increasing probesize\n", i);
        break;
    }
//注意, 如果文件中没有 Header (AVFMTCTX_NOHEADER), 可以在新的流中加入 Header
ret = read_frame_internal(ic, &pkt1); //步骤 3
if (ret == AVERERROR(EAGAIN))
    continue;
if (ret < 0) {
    //EOF 或者错误
    eof_reached = 1;
    break;
}
pkt = &pkt1;
if (!(ic->flags & AVFMT_FLAG_NOBUFFER)) {
    ret = add_to_pktbuf(&ic->internal->packet_buffer, pkt,
        &ic->internal->packet_buffer_end, 0);

    if (ret < 0)
        goto find_stream_info_err;
}
st = ic->streams[pkt->stream_index];
if (!(st->disposition & AV_DISPOSITION_ATTACHED_PIC))
    read_size += pkt->size;
avctx = st->internal->avctx;
if (!st->internal->avctx_inited) {
    ret = avcodec_parameters_to_context(avctx, st->codecpars);
    if (ret < 0)
        goto find_stream_info_err;
    st->internal->avctx_inited = 1;
}
if (pkt->dts != AV_NOPTS_VALUE && st->codec_info_nb_frames > 1) {
    /*检查没有增加的 DTS*/
    if (st->info->fps_last_dts != AV_NOPTS_VALUE &&
        st->info->fps_last_dts >= pkt->dts) {
        av_log(ic, AV_LOG_DEBUG,
            "Non-increasing DTS in stream %d: packet %d with DTS "
            "%PRId64", packet %d with DTS %PRId64\n",
            st->index, st->info->fps_last_dts_idx,
            st->info->fps_last_dts, st->codec_info_nb_frames,
```




```
        pkt->pts);
        st->info->fps_first_pts =
        st->info->fps_last_pts = AV_NOPTS_VALUE;
    }
    //检查PTS 是否不连续 (discontinuity)。如果PTS 的差异大于序列中平均
    //数据包持续时间的1000 倍，我们将其视为不连续的
    if (st->info->fps_last_pts != AV_NOPTS_VALUE &&
        st->info->fps_last_pts_idx > st->info->fps_first_pts_idx &&
        (pkt->pts - st->info->fps_last_pts) / 1000 >
        (st->info->fps_last_pts - st->info->fps_first_pts) /
idx)) {

        av_log(ic, AV_LOG_WARNING,
                "PTS discontinuity in stream %d: packet %d with PTS "
                "%\"PRIu64\", packet %d with PTS \"%\"PRIu64\"\n",
                st->index, st->info->fps_last_pts_idx,
                st->info->fps_last_pts, st->codec_info_nb_frames,
                pkt->pts);
        st->info->fps_first_pts =
        st->info->fps_last_pts = AV_NOPTS_VALUE;
    }
    /*更新存储的PTS 值*/
    if (st->info->fps_first_pts == AV_NOPTS_VALUE) {
        st->info->fps_first_pts = pkt->pts;
        st->info->fps_first_pts_idx = st->codec_info_nb_frames;
    }
    st->info->fps_last_pts = pkt->pts;
    st->info->fps_last_pts_idx = st->codec_info_nb_frames;
}
if (st->codec_info_nb_frames>1) {
    int64_t t = 0;
    int64_t limit;
    if (st->time_base.den > 0)
        t = av_rescale_q(st->info->codec_info_duration, st-> time_
base, AV_TIME_BASE_Q);
    if (st->avg_frame_rate.num > 0)
        t = FFMAX(t, av_rescale_q(st->codec_info_nb_frames, av_
inv_q(st->avg_frame_rate), AV_TIME_BASE_Q));
    if ( t == 0
        && st->codec_info_nb_frames>30
        && st->info->fps_first_pts != AV_NOPTS_VALUE
        && st->info->fps_last_pts != AV_NOPTS_VALUE)
        t = FFMAX(t, av_rescale_q(st->info->fps_last_pts - st->
info->fps_first_pts, st->time_base, AV_TIME_BASE_Q));
```

```
        if (analyzed_all_streams) limit = max_analyze_duration;
        else if (avctx->codec_type == AVMEDIA_TYPE_SUBTITLE) limit =
max_subtitle_analyze_duration;
        else
            limit = max_stream_analyze_duration;
        if (t >= limit) {
            av_log(ic, AV_LOG_VERBOSE, "max_analyze_duration %"PRIu64"
reached at %"PRIu64" microseconds st:%d\n",
                limit,
                t, pkt->stream_index);
            if (ic->flags & AVFMT_FLAG_NOBUFFER)
                av_packet_unref(pkt);
            break;
        }
        if (pkt->duration) {
            if (avctx->codec_type == AVMEDIA_TYPE_SUBTITLE && pkt-
>pts != AV_NOPTS_VALUE && pkt->pts >= st->start_time) {
                st->info->codec_info_duration = FFMIN(pkt->pts - st-
>start_time, st->info->codec_info_duration + pkt->duration);
            } else
                st->info->codec_info_duration += pkt->duration;
            st->info->codec_info_duration_fields += st->parser &&
st->need_parsing && avctx->ticks_per_frame == 2 ? st->parser->repeat_pict +
1 : 2;
        }
    }
    #if FF_API_R_FRAME_RATE
        if (st->codecpars->codec_type == AVMEDIA_TYPE_VIDEO)
            ff_rfps_add_frame(ic, st, pkt->dts);
    #endif

    if (st->parser && st->parser->parser->split && !avctx->extradata) {
        int i = st->parser->parser->split(avctx, pkt->data, pkt-
>size);

        if (i > 0 && i < FF_MAX_EXTRADATA_SIZE) {
            avctx->extradata_size = i;
            avctx->extradata = av_mallocz(avctx->extradata_size +
                AV_INPUT_BUFFER_PADDING_SIZE);
            if (!avctx->extradata)
                return AERROR(ENOMEM);
            memcpy(avctx->extradata, pkt->data,
                avctx->extradata_size);
        }
    }
    try_decode_frame(ic, st, pkt,
        (options && i < orig_nb_streams) ? &options[i] : NULL);
```



```
if (ic->flags & AVFMT_FLAG_NOBUFFER)
    av_packet_unref(pkt);
st->codec_info_nb_frames++;
count++;
}
```

这个循环代码很长，主要是将数据读进来并解码分析数据流，拆分步骤主要如下。

(1) 检查用户有没有请求中断。如果有请求中断，就调用中断回调方法，并且函数返回。

(2) 再一次遍历流，检查是不是还有 Codec 需要进一步处理。这里会检查 Codec 的各个参数，如果这些参数都已经被初始化过，那么就会返回，也就没有必要再做进一步的分析了。如果还有一些 Codec 的信息不全，那么这里会继续向下执行。

(3) 读 1 帧数据进来，调用的是 `read_frame_internal` 函数。这个函数将在后续章节中分析。

(4) 把读出来的数据添加到缓冲区，调用的是 `add_to_pktbuf` 函数。然后更新读到的总的
数据大小 `read_size += pkt->size;`。

(5) 再次更新 Codec 上下文环境的参数，调用的是 `avcodec_parameters_to_context` 函数。

(6) 检查 DTS 是否 `discontinuely`（不连续）。在 HLS 协议中，也有这个标签，这个标签会导致原生播放器重启，表现得有点卡顿，在 FFmpeg 中，同样也要检测 DTS 是否 `discontinuely`（不连续），以便做 DTS 累加操作。

(7) 更新 DTS 的值。

(8) 如果仍然没有播放器开始播放所需要的信息，我们尝试打开编解码器并且解压缩帧。在通常情况下，要避免走到这一步，一旦使用解码器获取更多信息，就会非常耗时，并且要占用更多的内存，而实际上还没到解码步骤。如果实在没有起播所需要的信息，就要强制地让解码器解码至少 1 帧数据出来，获取里面的信息。我们在大多数情况下需要尽量避免做这样的事情，因为这将花费大量的时间，并且使用更多的内存。尝试解压 1 帧数据使用的是 `try_decode_frame` 函数。有些参数，如视频的 `pix_fmt`，需要调用 `h264_decode_frame` 函数才可以获取。

最后总结一下，这个循环体主要会检测是不是所有流信息都具备起播条件，如果具备就直接返回，否则就尝试解码至少 1 帧数据，并从中获取 `pix_fmt` 等必须通过解码 1 帧数据才能获得的参数。接下来是判断是否到达数据流文件结束位置的逻辑，代码如下：

```
if (eof_reached) {
    int stream_index;
    for (stream_index = 0; stream_index < ic->nb_streams; stream_index++)
```




```
index++) {
    st = ic->streams[stream_index];
    avctx = st->internal->avctx;
    if (!has_codec_parameters(st, NULL)) {
        const AVCodec *codec = find_decoder(ic, st, st-> codecpar
-> codec_id);
        if (codec && !avctx->codec) {
            //步骤 2
            if (avcodec_open2(avctx, codec, (options && stream_
index < orig_nb_streams) ? &options[stream_index] : NULL) < 0)
                av_log(ic, AV_LOG_WARNING,
                    "Failed to open codec in av_find_stream_info\n");
        }
    }
    //当正在读流时, EOF 已经到达, 所以无论延时多久, 都需要对 DTS 重新排序
    if (ic->internal->packet_buffer && !has_decode_delay_been_
guessed(st)) {
        update_dts_from_pts(ic, stream_index, ic->internal-> packet_
buffer);
    }
}
}
```

检查是否已经到达文件尾部, 如果到达文件尾部, 又会遍历一次数据流, 检测其 Codec 的参数, 如果参数不完整, 就会再次调用 `avcodec_open2` 函数来初始化编解码器的各个参数。接下来判断是否刷新解码器, 代码如下:

```
if (flush_codecs) {
    AVPacket empty_pkt = { 0 };
    int err = 0;
    av_init_packet(&empty_pkt);
    for (i = 0; i < ic->nb_streams; i++) {
        st = ic->streams[i];
        /*刷新解码器, 清掉缓冲数据, 重新解码*/
        if (st->info->found_decoder == 1) {
            do {
                //步骤 4
                err = try_decode_frame(ic, st, &empty_pkt,
                    (options && i < orig_nb_streams)
                    ? &options[i] : NULL);
            } while (err > 0 && !has_codec_parameters(st, NULL));
            if (err < 0) {
                av_log(ic, AV_LOG_INFO,
                    "decoding for stream %d failed\n", st->index);
            }
        }
    }
}
```




```
    }  
    }  
}
```

刷新缓冲区，输出解码器中缓存的帧，刷新之后，看看后面的代码：

```
//调用 try_decode_frame 函数时，关闭之前的编解码器  
for (i = 0; i < ic->nb_streams; i++) {  
    st = ic->streams[i];  
    avcodec_close(st->internal->avctx);  
}
```

遍历数据流，调用 `avcodec_close` 函数关闭编解码器。接下来的逻辑开始处理音频、视频流，代码如下：

```
ff_rfps_calculate(ic);  
for (i = 0; i < ic->nb_streams; i++) {  
    st = ic->streams[i];  
    avctx = st->internal->avctx;  
    if (avctx->codec_type == AVMEDIA_TYPE_VIDEO) {  
        if (avctx->codec_id == AV_CODEC_ID_RAWVIDEO && !avctx-> codec_  
tag && !avctx->bits_per_coded_sample) {  
            uint32_t tag= avcodec_pix_fmt_to_codec_tag(avctx-> pix_  
fmt);  
            if (avpriv_find_pix_fmt(avpriv_get_raw_pix_fmt_tags(), tag)  
== avctx->pix_fmt)  
                avctx->codec_tag= tag;  
        }  
        //如果解复用器没有设置帧率，那么需要估计一个平均帧率  
        if (st->info->codec_info_duration_fields &&  
!st->avg_frame_rate.num &&  
st->info->codec_info_duration) {  
            int best_fps = 0;  
            double best_error = 0.01;  
            if (st->info->codec_info_duration >= INT64_MAX /  
st->time_base.num / 2 || st->info->codec_info_duration_fields >= INT64_MAX /  
st->time_base.den || st->info->codec_info_duration < 0)  
                continue;  
            av_reduce(&st->avg_frame_rate.num, &st->avg_frame_rate.den,  
st->info->codec_info_duration_fields * (int64_t)  
st->time_base.den,  
st->info->codec_info_duration * 2 * (int64_t)  
st->time_base.num, 60000);  
            //如果在原始估计的 1%以内，圆形估计的帧率为一个“标准”帧率  
            for (j = 0; j < MAX_STD_TIMEBASES; j++) {  
                AVRational std_fps = { get_std_framerate(j), 12 * 1001 };
```



```
double error = fabs(av_q2d(st->avg_frame_rate) /
                    av_q2d(std_fps) - 1);

    if (error < best_error) {
        best_error = error;
        best_fps = std_fps.num;
    }
}
if (best_fps)
    av_reduce(&st->avg_frame_rate.num, &st-> avg_frame_
rate.den, best_fps, 12 * 1001, INT_MAX);
}
if (!st->r_frame_rate.num) {
    if ( avctx->time_base.den * (int64_t) st-> time_base.
num <= avctx->time_base.num * avctx->ticks_per_frame * (int64_t) st-> time_
base.den) {
        st->r_frame_rate.num = avctx->time_base.den;
        st->r_frame_rate.den = avctx->time_base.num*avctx->
ticks_per_frame;
    } else {
        st->r_frame_rate.num = st->time_base.den;
        st->r_frame_rate.den = st->time_base.num;
    }
}
if (st->display_aspect_ratio.num && st-> display_aspect_
ratio.den) {
    AVRational hw_ratio = { avctx->height, avctx->width };
    st->sample_aspect_ratio = av_mul_q(st-> display_ aspect_
ratio, hw_ratio);
}
} else if (avctx->codec_type == AVMEDIA_TYPE_AUDIO) {
    if (!avctx->bits_per_coded_sample)
        avctx->bits_per_coded_sample =
            av_get_bits_per_sample(avctx->codec_id);
    //根据音频服务类型设置流配置
    switch (avctx->audio_service_type) {
    case AV_AUDIO_SERVICE_TYPE_EFFECTS:
        st->disposition = AV_DISPOSITION_CLEAN_EFFECTS;
        break;
    case AV_AUDIO_SERVICE_TYPE_VISUALLY_IMPAIRED:
        st->disposition = AV_DISPOSITION_VISUAL_IMPAIRED;
        break;
    case AV_AUDIO_SERVICE_TYPE_HEARING_IMPAIRED:
        st->disposition = AV_DISPOSITION_HEARING_IMPAIRED;
```




```
        break;
    case AV_AUDIO_SERVICE_TYPE_COMMENTARY:
        st->disposition = AV_DISPOSITION_COMMENT;
        break;
    case AV_AUDIO_SERVICE_TYPE_KARAOKE:
        st->disposition = AV_DISPOSITION_KARAOKE;
        break;
    }
}
```

这个循环用来处理音频流和视频流，最后剩余的代码如下：

```
if (probesize)
    estimate_timings(ic, old_offset);
av_opt_set(ic, "skip_clear", "0", AV_OPT_SEARCH_CHILDREN);
if (ret >= 0 && ic->nb_streams)
    //在 EOF 之前，我们不可能有所有的编解码器参数
    ret = -1;
for (i = 0; i < ic->nb_streams; i++) {
    const char *errmsg;
    st = ic->streams[i];
    //如果没有等到新的 Packet，更新 has_codec_parameters 的上下文
    if (!st->internal->avctx_inited) {
        if (st->codecpa->codec_type == AVMEDIA_TYPE_AUDIO &&
            st->codecpa->format == AV_SAMPLE_FMT_NONE)
            st->codecpa->format = st->internal->avctx->sample_fmt;
        ret = avcodec_parameters_to_context(st->internal->avctx, st->codecpa);
        if (ret < 0)
            goto find_stream_info_err;
    }
    if (!has_codec_parameters(st, &errmsg)) {
        char buf[256];
        avcodec_string(buf, sizeof(buf), st->internal->avctx, 0);
        av_log(ic, AV_LOG_WARNING,
            "Could not find codec parameters for stream %d (%s): %s\n"
            "Consider increasing the value for the 'analyzeduration'
and 'probesize' options\n", i, buf, errmsg);
    } else {
        ret = 0;
    }
}
compute_chapters_end(ic);
/*从内部编解码上下文环境更新流的参数*/
```



```

for (i = 0; i < ic->nb_streams; i++) {
    st = ic->streams[i];
    if (st->internal->avctx_inited) {
        int orig_w = st->codecpar->width;
        int orig_h = st->codecpar->height;
        ret = avcodec_parameters_from_context(st->codecpar, st->
internal->avctx);
        if (ret < 0)
            goto find_stream_info_err;
        //解码器可以降低视频大小的低分辨率影响
        if (av_codec_get_lowres(st->internal->avctx) && orig_w) {
            st->codecpar->width = orig_w;
            st->codecpar->height = orig_h;
        }
    }
    #if FF_API_LAVF_AVCTX
    FF_DISABLE_DEPRECATION_WARNINGS
    ret = avcodec_parameters_to_context(st->codec, st->codecpar);
    if (ret < 0)
        goto find_stream_info_err;
    //旧的 API (AVStream.codec) “要求” 分辨率要根据低分辨率系数进行调整
    if (av_codec_get_lowres(st->internal->avctx) && st->internal->
avctx->width) {
        av_codec_set_lowres(st->codec, av_codec_get_lowres(st->internal->
avctx));

        st->codec->width = st->internal->avctx->width;
        st->codec->height = st->internal->avctx->height;
    }
    if (st->codec->codec_tag != MKTAG('t', 'm', 'c', 'd'))
        st->codec->time_base = st->internal->avctx->time_base;
    st->codec->framerate = st->avg_frame_rate;
    if (st->internal->avctx->subtitle_header) {
        st->codec->subtitle_header = av_malloc(st->internal->avctx->
subtitle_header_size);
        if (!st->codec->subtitle_header)
            goto find_stream_info_err;
        st->codec->subtitle_header_size = st->internal->avctx-> subtitle_
header_size;
        memcpy(st->codec->subtitle_header, st->internal->avctx-> subtitle_
header, st->codec->subtitle_header_size);
    }

    //AVCodecParameters 中的成员变量不可用
    st->codec->coded_width = st->internal->avctx->coded_width;

```

```

        st->codec->coded_height = st->internal->avctx->coded_height;
        st->codec->properties = st->internal->avctx->properties;
FF_ENABLE_DEPRECATION_WARNINGS
#endif
        st->internal->avctx_inited = 0;
    }
find_stream_info_err:
    for (i = 0; i < ic->nb_streams; i++) {
        st = ic->streams[i];
        if (st->info)
            av_freep(&st->info->duration_error);
        av_freep(&ic->streams[i]->info);
    }
    if (ic->pb)
        av_log(ic, AV_LOG_DEBUG, "After avformat_find_stream_info() pos:
%"PRId64" bytes read:%"PRId64" seeks:%d frames:%d\n", avio_tell(ic->pb), ic-
>pb->bytes_read, ic->pb->seek_count, count);
    return ret;
}

```

最后就是更新各个结构的数据了。AVStream 中的 AVCodecContext 结构体，以及 AVStream 中 AVStreamInternal 的 AVCodecContext 结构体的数据要统一。

8.2.6 av_read_frame 函数

av_read_frame 函数的定义位于 libavformat/utils.c，其代码如下：

```

int av_read_frame(AVFormatContext *s, AVPacket *pkt)
{
    const int genpts = s->flags & AVFMT_FLAG_GENPTS;
    int eof = 0;
    int ret;
    AVStream *st;
    if (!genpts) {
        //从音视频压缩数据的缓冲 Buffer 中读取
        ret = s->internal->packet_buffer
            ? read_from_packet_buffer(&s->internal->packet_buffer,
                                     &s->internal->packet_buffer_end, pkt)
            : read_frame_internal(s, pkt);
        if (ret < 0)
            return ret;
        goto return_packet;
    }
    for (;;) {
        AVPacketList *pkt1 = s->internal->packet_buffer;

```



```

    if (pkt1) {
        AVPacket *next_pkt = &pkt1->pkt;
        if (next_pkt->dts != AV_NOPTS_VALUE) {
            int wrap_bits = s->streams[next_pkt->stream_index]-> pts_
wrap_bits;

            //当前数据流中出现过 DTS, 但是当前 Packet 后面没有任何 Packet 包含
            //DTS, 需要设置这个 Packet 为 AV_NOPTS_VALUE 属性
            int64_t last_dts = next_pkt->dts;
            while (pkt1 && next_pkt->pts == AV_NOPTS_VALUE) {
                if (pkt1->pkt.stream_index == next_pkt->stream_index &&
(av_compare_mod(next_pkt->dts, pkt1->pkt.dts, 2LL << (wrap_bits - 1)) < 0)) {
                    if (av_compare_mod(pkt1->pkt.pts, pkt1->pkt.dts, 2LL
<< (wrap_bits - 1))) {

                        next_pkt->pts = pkt1->pkt.dts; //不是 B 帧
                    }
                    if (last_dts != AV_NOPTS_VALUE) {
                        //一旦最后一个 DTS 被设置成 AV_NOPTS_VALUE, 我们就不能改变了
                        //当 DTS 不为无效值时, 可进行赋值
                        last_dts = pkt1->pkt.dts;
                    }
                }
                pkt1 = pkt1->next;
            }
            if (eof && next_pkt->pts == AV_NOPTS_VALUE && last_dts !=
AV_NOPTS_VALUE) {
                //修复最后一个参考帧没有 PTS 问题
                next_pkt->pts = last_dts + next_pkt->duration;
            }
            pkt1 = s->internal->packet_buffer;
        }
        //从音视频压缩数据缓冲 Buffer 中读取压缩数据
        st = s->streams[next_pkt->stream_index];
        if (!(next_pkt->pts == AV_NOPTS_VALUE && st->discard <
AVDISCARD_ALL && next_pkt->dts != AV_NOPTS_VALUE && !eof)) {
            ret = read_from_packet_buffer(&s->internal-> packet_ buffer,
&s->internal->packet_buffer_end, pkt);
            goto return_packet;
        }
    }
    ret = read_frame_internal(s, pkt);
    if (ret < 0) {
        if (pkt1 && ret != AERROR(EAGAIN)) {
            eof = 1;

```



```

        continue;
    } else
        return ret;
}
ret = add_to_pktbuf(&s->internal->packet_buffer, pkt,
                  &s->internal->packet_buffer_end, 1);
av_packet_unref(pkt);
if (ret < 0)
    return ret;
}
return_packet:
    st = s->streams[pkt->stream_index];
    if ((s->iformat->flags & AVFMT_GENERIC_INDEX) && pkt->flags & AV_
PKT_FLAG_KEY) {
        ff_reduce_index(s, st->index);
        av_add_index_entry(st, pkt->pos, pkt->pts, 0, 0, AVINDEX_KEYFRAME);
    }
    if (is_relative(pkt->pts))
        pkt->pts -= RELATIVE_TS_BASE;
    if (is_relative(pkt->pts))
        pkt->pts -= RELATIVE_TS_BASE;
    return ret;
}

```

在上面的代码中，`av_read_packet` 函数读出的是包，其可能是半帧或多帧，不保证帧的完整性。`av_read_frame` 对 `av_read_packet` 进行了封装，使读出的数据总是完整的帧。`av_read_frame` 函数调用了 `read_frame_internal`。

FFmpeg 中的 `av_read_frame` 函数的作用是读取码流中的若干音频帧或者 1 帧视频。例如，在解码视频的时候，每解码一个视频帧，需要先调用 `av_read_frame` 获得 1 帧视频的压缩数据，然后才能对该数据进行解码（例如，H.264 中 1 帧压缩数据通常对应一个 NAL）。

调用 `av_read_frame` 函数有如下 3 种情况。

(1) 如果 `packet_buffer` 存在数据，根据 PTS 返回 AVPacket。

(2) 如果 `packet_buffer` 不存在数据，调用 `av_read_frame_internal` 函数。

(3) 在 FFmpeg 中实现了将封装格式文件的 Packet 最终通过解码转换成解码后的帧，并解析填充了音视频压缩数据的 PTS、DTS 等信息，为最终解码提供了重要的数据，`av_read_frame_internal` 调用 `av_read_packet`，每次只读取一个包，然后直到解析完这个包的所有数据，才开始读取下一个包，解析完的数据被保存在 `parser`（解析器）结构的数据缓冲区中，这样即使 `av_read_packet` 读取的下一个包和前一个包的流不一样，但由于 `parser` 也不一样，所以实现

了 `av_read_frame_internal` 函数调用可以解析出不同流的原始数据流（俗称裸流），而 `av_read_frame_internal` 函数除非出错，否则必须解析出 1 帧数据才能返回。

调用 `av_parser_parse2` 函数分析出视频 1 帧（或音频若干帧），下面看看 `read_frame_internal` 函数：

```
static int read_frame_internal(AVFormatContext *s, AVPacket *pkt)
{
    int ret = 0, i, got_packet = 0;
    AVDictionary *metadata = NULL;
    av_init_packet(pkt);
    while (!got_packet && !s->internal->parse_queue) {
        AVStream *st;
        AVPacket cur_pkt;

        ret = ff_read_packet(s, &cur_pkt); //读取下一个音视频压缩数据
        if (ret < 0) {
            if (ret == AVERERROR(EAGAIN))
                return ret;
            //刷新解析器
            for (i = 0; i < s->nb_streams; i++) { //遍历刷新解析器
                st = s->streams[i];
                if (st->parser && st->need_parsing)
                    parse_packet(s, NULL, st->index);
            }
            /*所有剩余的 Packet 都在 parse_queue 中=>
            *真正地终止了解析*/
            break;
        }
        ret = 0;
        st = s->streams[cur_pkt.stream_index];
        //省略部分代码
        if (!st->need_parsing || !st->parser) {
            /*不需要解析，按原样输出数据包*/
            *pkt = cur_pkt;
            compute_pkt_fields(s, st, NULL, pkt, AV_NOPTS_VALUE, AV_
NOPTS_VALUE);
            if ((s->iformat->flags & AVFMT_GENERIC_INDEX) &&
                (pkt->flags & AV_PKT_FLAG_KEY) && pkt->pts != AV_NOPTS_
VALUE) {
                ff_reduce_index(s, st->index);
                av_add_index_entry(st, pkt->pos, pkt->pts,
                                0, 0, AVINDEX_KEYFRAME);
            }
        }
    }
}
```



```

    }
    got_packet = 1;
} else if (st->discard < AVDISCARD_ALL) {
    if ((ret = parse_packet(s, &cur_pkt, cur_pkt.stream_index)) < 0)
        return ret;
    st->codecpars->sample_rate = st->internal->avctx->sample_rate;
    st->codecpars->bit_rate = st->internal->avctx->bit_rate;
    st->codecpars->channels = st->internal->avctx->channels;
    st->codecpars->channel_layout = st->internal->avctx->channel_
layout;
    st->codecpars->codec_id = st->internal->avctx->codec_id;
} else {
    /* 释放音视频压缩数据 */
    av_packet_unref(&cur_pkt);
}
if (pkt->flags & AV_PKT_FLAG_KEY)
    st->skip_to_keyframe = 0;
if (st->skip_to_keyframe) {
    av_packet_unref(&cur_pkt);
    if (got_packet) {
        *pkt = cur_pkt;
    }
    got_packet = 0;
}
}

if (!got_packet && s->internal->parse_queue)
    ret = read_from_packet_buffer(&s->internal->parse_queue, &s->
internal->parse_queue_end, pkt);

if (ret >= 0) {
    AVStream *st = s->streams[pkt->stream_index];
    int discard_padding = 0;
    if (st->first_discard_sample && pkt->pts != AV_NOPTS_VALUE) {
        int64_t pts = pkt->pts - (is_relative(pkt->pts) ? RELATIVE_
TS_BASE : 0);
        int64_t sample = ts_to_samples(st, pts);
        int duration = ts_to_samples(st, pkt->duration);
        int64_t end_sample = sample + duration;
        if (duration > 0 && end_sample >= st->first_discard_sample
&&
            sample < st->last_discard_sample)
            discard_padding = FFMIN(end_sample - st->first_discard_
sample, duration);
    }
}

```



```

    }
    if (st->start_skip_samples && (pkt->pts == 0 || pkt->pts == RELATIVE_
TS_BASE))
        st->skip_samples = st->start_skip_samples;
    if (st->skip_samples || discard_padding) {
        uint8_t *p = av_packet_new_side_data(pkt, AV_PKT_DATA_SKIP_
SAMPLES, 10);
        if (p) {
            AV_WL32(p, st->skip_samples);
            AV_WL32(p + 4, discard_padding);
            av_log(s, AV_LOG_DEBUG, "demuxer injecting skip %d /
discard %d\n", st->skip_samples, discard_padding);
        }
        st->skip_samples = 0;
    }

    if (!(s->flags & AVFMT_FLAG_KEEP_SIDE_DATA))
        av_packet_merge_side_data(pkt);
}
av_opt_get_dict_val(s, "metadata", AV_OPT_SEARCH_CHILDREN, &metadata);
//省略部分代码
return ret;
}

```

`read_frame_internal` 函数的代码比较长，这里只简单看一下它前面的部分。它前面的部分有两步是十分关键的：

(1) 调用了 `ff_read_packet` 函数，从对应的 `AVInputFormat` 读取数据。

(2) 如果媒体视频流/音频流需要使用 `AVCodecParser`，则调用 `parse_packet` 函数解析相应的 `AVPacket`。

下面看一下 `ff_read_packet` 函数和 `parse_packet` 函数的源代码：

```

int ff_read_packet(AVFormatContext *s, AVPacket *pkt)
{
    int ret, i, err;
    AVStream *st;
    for (;;) {
        AVPacketList *pktl = s->internal->raw_packet_buffer;
        if (pktl) {
            *pkt = pktl->pkt;
            st = s->streams[pkt->stream_index];
            if (s->internal->raw_packet_buffer_remaining_size <= 0)
                if ((err = probe_codec(s, st, NULL)) < 0)

```

```

        return err;
    if (st->request_probe <= 0) {
        s->internal->raw_packet_buffer = pkt1->next;
        s->internal->raw_packet_buffer_remaining_size += pkt->size;
        av_free(pkt1);
        return 0;
    }
    pkt->data = NULL;
    pkt->size = 0;
    av_init_packet(pkt);
    ret = s->iformat->read_packet(s, pkt);
    //省略部分代码
    if (!pkt->buf) {
        AVPacket tmp = { 0 };
        ret = av_packet_ref(&tmp, pkt);
        if (ret < 0)
            return ret;
        *pkt = tmp;
    }
    st = s->streams[pkt->stream_index];
    //省略部分代码
    err = add_to_pktbuf(&s->internal->raw_packet_buffer, pkt,
                      &s->internal->raw_packet_buffer_end, 0);
    if (err)
        return err;
    s->internal->raw_packet_buffer_remaining_size -= pkt->size;
    if ((err = probe_codec(s, st, pkt)) < 0)
        return err;
}
}

```

ff_read_packet 函数中最关键的地方就是，调用了 AVInputFormat 的 read_packet 函数。AVInputFormat 的 read_packet 是一个函数指针，指向当前 AVInputFormat 读取数据的函数。

8.2.7 av_write_frame 函数

FFmpeg 调用 avformat_write_header 函数写头部信息，调用 av_write_frame 函数写 1 帧数据，调用 av_write_trailer 函数写尾部信息，它们都使用 AVFormatContext 结构体作为参数，这些代码如下：

```

int av_write_frame(AVFormatContext *s, AVPacket *pkt)
{
    int ret;

```



```

ret = prepare_input_packet(s, pkt); //1
if (ret < 0)
    return ret;
if (!pkt) {
    if (s->oformat->flags & AVFMT_ALLOW_FLUSH) {
        if (!s->internal->header_written) {
            ret = s->internal->write_header_ret ? s->internal->write_
header_ret : write_header_internal(s);
            if (ret < 0)
                return ret;
        }
        ret = s->oformat->write_packet(s, NULL); //2
        if (s->flush_packets && s->pb && s->pb->error >= 0 && s->
flags & AVFMT_FLAG_FLUSH_PACKETS)
            avio_flush(s->pb); //3
        if (ret >= 0 && s->pb && s->pb->error < 0)
            ret = s->pb->error;
        return ret;
    }
    return 1;
}
#endif
ret = compute_muxer_pkt_fields(s, s->streams[pkt->stream_index], pkt);
if (ret < 0 && !(s->oformat->flags & AVFMT_NOTIMESTAMPS))
    return ret;
#endif
ret = write_packet(s, pkt); //4
if (ret >= 0 && s->pb && s->pb->error < 0)
    ret = s->pb->error;
if (ret >= 0)
    s->streams[pkt->stream_index]->nb_frames++; //对应的流加 1 帧
return ret;
}

```

简单解释一下 `av_write_frame` 函数中它的参数的含义。

- `s`: 用于输出的 `AVFormatContext`。
- `pkt`: 等待输出的 `AVPacket`。

函数正常执行后返回值等于 0。

从源代码可以看出, `av_write_frame` 函数主要完成了以下几步工作。

- (1) 调用 `prepare_input_packet` 函数做一些简单的检测。
- (2) 调用 `avio_flush` 函数设置 `AVPacket` 的一些属性值。

Android 音视频开发

(3) 调用 `write_packet` 函数写入数据。

下面通过代码看一下这几个函数的功能：

```
static int prepare_input_packet(AVFormatContext *s, AVPacket *pkt)
{
    int ret;
    ret = check_packet(s, pkt); //做一些检测
    if (ret < 0)
        return ret;
#ifdef FF_API_COMPUTE_PKT_FIELDS2 && FF_API_LAVF_AVCTX
    /*检查时间戳*/
    if (!(s->oformat->flags & AVFMT_NOTIMESTAMPS)) {
        AVStream *st = s->streams[pkt->stream_index]; //
        //在没有重新排序的时候（所以 DTS 等于 PTS）
        if (!st->internal->reorder) {
            if (pkt->pts == AV_NOPTS_VALUE && pkt->dts != AV_NOPTS_VALUE)
                pkt->pts = pkt->dts;
            if (pkt->dts == AV_NOPTS_VALUE && pkt->pts != AV_NOPTS_VALUE)
                pkt->dts = pkt->pts;
        }
        //检查是否设置了时间戳
        if (pkt->pts == AV_NOPTS_VALUE || pkt->dts == AV_NOPTS_VALUE) {
            av_log(s, AV_LOG_ERROR,
                "Timestamps are unset in a packet for stream %d\n",
st->index);
            return AERROR(EINVAL);
        }
        //如果格式支持，检查 DTS 是否增加
        if (st->cur_dts != AV_NOPTS_VALUE &&
            ((!(s->oformat->flags & AVFMT_TS_NONSTRICT) && st->cur_dts
            >= pkt->dts) || st->cur_dts > pkt->dts)) {
            av_log(s, AV_LOG_ERROR, "Application provided invalid, non
monotonically increasing "
                "dts to muxer in stream %d: %" PRId64 " >= %" PRId64 "
"\n", st->index, st->cur_dts, pkt->dts);
            return AERROR(EINVAL);
        }
        if (pkt->pts < pkt->dts) {
            av_log(s, AV_LOG_ERROR, "pts %" PRId64 " < dts %" PRId64 "
in stream %d\n", pkt->pts, pkt->dts, st->index);
            return AERROR(EINVAL);
        }
    }
}
#endif
```

```
    return 0;
}
```

FFmpeg 的复用操作主要分为 3 步:

- (1) `avformat_write_header`: 写文件头。
- (2) `av_write_frame/av_interleaved_write_frame`: 写 Packet。
- (3) `av_write_trailer`: 写文件尾。

第一步写文件头, 主要是把音视频的一些基本信息放入头部, 不同格式的音视频头部信息不一样。 `write_header_internal` (写文件头) 函数的代码如下:

```
static int write_header_internal(AVFormatContext *s)
{
    if (!(s->oformat->flags & AVFMT_NOFILE) && s->pb)
        avio_write_marker(s->pb, AV_NOPTS_VALUE, AVIO_DATA_MARKER_HEADER);
    if (s->oformat->write_header) {
        int ret = s->oformat->write_header(s);
        //outputformatContext 写入 header
        if (ret >= 0 && s->pb && s->pb->error < 0)
            ret = s->pb->error;
        s->internal->write_header_ret = ret;
        if (ret < 0)
            return ret;
        if (s->flush_packets && s->pb && s->pb->error >= 0 && s->flags &
AVFMT_FLAG_FLUSH_PACKETS)
            avio_flush(s->pb); //刷新缓冲区
    }
    s->internal->header_written = 1;
    if (!(s->oformat->flags & AVFMT_NOFILE) && s->pb)
        avio_write_marker(s->pb, AV_NOPTS_VALUE, AVIO_DATA_MARKER_UNKNOWN);
    return 0;
}
```

可见 `write_header_internal` 直接调用了匹配的 `AVOutputFormat` 的 `write_header` 函数。因此, 具体的 `write_header` 函数由各种封装格式自己定义并实现。写入一个音视频压缩数据。如果在标志中设置了 `AVFMT_ALLOW_FLUSH`, 则 `PKT` 可以为空, 以便在 `Muxer` 中刷新数据。当刷新数据时, 如果仍然有很多的数据需要刷新, 则返回 0; 如果所有的数据都被刷新了, 则没有更多的缓冲数据。写完文件头中的数据后, 开始写 `Packet`, 主要就是音视频数据, 代码如下:

```
static int write_packet(AVFormatContext *s, AVPacket *pkt)
{

```


Android 音视频开发

```
char buf[256];
if (pkt->stream_index)
    av_log(s, AV_LOG_WARNING, "More than one stream unsupported\n");
snprintf(buf, sizeof(buf), "%" PRIu64 "\n", pkt->pts);
avio_write(s->pb, buf, strlen(buf));
return 0;
}
```

实际上就是把 PKT 的 DTS 格式化到 buf 中，然后通过 avio_write 写到队列中。最后一步就是写文件尾，标示一个 Packet 完整结束。缓冲 Buffer 中又开始写下一个 Packet，这样循环往复，直到把 Packet 写完，FFmpeg 复用操作就完成了。

8.2.8 avcodec_decode_video2 函数

avcodec_decode_video2 函数将解码 AVPacket 的数据，并将解码后的数据填充到 AVFrame 中，AVFrame 中保存的是解码后的原始数据。avcodec_decode_video2 函数的代码如下：

```
int attribute_align_arg avcodec_decode_video2(AVCodecContext *avctx,
AVFrame *picture, int *got_picture_ptr, const AVPacket *avpkt)
{
    AVCodecInternal *avci = avctx->internal;
    int ret;
    //复制一个副本，确保我们不改变 avpkt 指针变量中的内容
    AVPacket tmp = *avpkt;
    if (!avctx->codec)
        return AVERERROR(EINVAL);
    if (avctx->codec->type != AVMEDIA_TYPE_VIDEO) {
        av_log(avctx, AV_LOG_ERROR, "Invalid media type for video\n");
        return AVERERROR(EINVAL);
    }
    if (!avctx->codec->decode) {
        av_log(avctx, AV_LOG_ERROR, "This decoder requires using the
avcodec_send_packet() API.\n");
        return AVERERROR(ENOSYS);
    }
    *got_picture_ptr = 0;
    if ((avctx->coded_width || avctx->coded_height) && av_image_check_
size(avctx->coded_width, avctx->coded_height, 0, avctx))
        return AVERERROR(EINVAL);
    avctx->internal->pkt = avpkt;
    ret = apply_param_change(avctx, avpkt);
    if (ret < 0)
        return ret;
    av_frame_unref(picture);
```


第8章 FFmpeg 源码分析及实战

```

if ((avctx->codec->capabilities & AV_CODEC_CAP_DELAY) || avpkt->size ||
    (avctx->active_thread_type & FF_THREAD_FRAME)) {
    int did_split = av_packet_split_side_data(&tmp);
    ret = apply_param_change(avctx, &tmp);
    if (ret < 0)
        goto fail;
    avctx->internal->pkt = &tmp;
    if (HAVE_THREADS && avctx->active_thread_type & FF_THREAD_FRAME)
        ret = ff_thread_decode_frame(avctx, picture, got_picture_ptr,
                                     &tmp); //多线程解码时, 送 frame 到解码线程中
    else {
        ret = avctx->codec->decode(avctx, picture, got_picture_ptr,
                                   &tmp); //若不是多线程, 就直接使用 AVCodec 的 decode 函数进行解码
        if (!(avctx->codec->caps_internal & FF_CODEC_CAP_SETS_PKT_DTS))
            picture->pkt_dts = avpkt->dts;
        if (!avctx->has_b_frames) {
            av_frame_set_pkt_pos(picture, avpkt->pos);
        }
        //应该在有 B 帧的判断下
        //get_buffer 是用来设置帧参数的
        if (!(avctx->codec->capabilities & AV_CODEC_CAP_DR1)) {
            if (!picture->sample_aspect_ratio.num) picture->sample_
aspect_ratio = avctx->sample_aspect_ratio;
            if (!picture->width) picture->width = avctx->width;
            if (!picture->height) picture->height = avctx->height;
            if (picture->format == AV_PIX_FMT_NONE) picture->format
= avctx->pix_fmt;
        }
    }
}

fail:
    emms_c(); //避免在每次返回前调用 emms_c
    avctx->internal->pkt = NULL;
    if (did_split) {
        av_packet_free_side_data(&tmp);
        if (ret == tmp.size)
            ret = avpkt->size;
    }
    if (*got_picture_ptr) {
        if (!avctx->refcounted_frames) {
            int err = unrefcount_frame(avci, picture);
            if (err < 0)
                return err;
        }
    }
}

```

Android 音视频开发

```

        avctx->frame_number++;
        av_frame_set_best_effort_timestamp(picture,
                                           guess_correct_pts(avctx,
                                           picture->pkt_pts,
                                           picture->pkt_dts));

    } else
        av_frame_unref(picture);
    } else
        ret = 0;
    //许多解码器分配整个 AVFrames 空间, 因此需要覆盖 extended_data, 以确保它被正确设置
    av_assert0(!picture->extended_data || picture->extended_data == picture
-> data);
    #if FF_API_AVCTX_TIMEBASE
        if (avctx->framerate.num > 0 && avctx->framerate.den > 0)
            avctx->time_base = av_inv_q(av_mul_q(avctx->framerate, (AVRational)
{avctx->ticks_per_frame, 1}));
    #endif
    return ret;
}

```

从代码中可以看出, `avcodec_decode_video2` 函数主要做了以下几方面的工作。

- (1) 对输入字段进行了一系列的检查工作, 例如, 宽高值是否正确, 输入是否为视频等。
- (2) 通过 `ret = avctx->codec->decode(avctx, picture, got_picture_ptr, &tmp)` 这句代码, 调用了相应的 AVCodec 的 `decode` 函数, 完成了解码操作。
- (3) 对得到的 AVFrame 的一些字段进行了赋值, 例如宽高值、像素格式等。

其中第 (2) 步是关键的一步, 它调用了 AVCodec 的 `decode` 函数完成了解码。AVCodec 的 `decode` 函数是一个函数指针, 指向了具体解码器的解码函数。在这里我们以 HEVC 解码器为例, 看看解码的实现过程。HEVC 解码器对应的 AVCodec 的定义位于 `libavcodec/hevc.c`, 代码如下:

```

AVCodec ff_hevc_decoder = {
    .name           = "hevc",
    .long_name      = NULL_IF_CONFIG_SMALL("HEVC (High Efficiency
Video Coding)"),
    .type           = AVMEDIA_TYPE_VIDEO,
    .id             = AV_CODEC_ID_HEVC,
    .priv_data_size = sizeof(HEVCContext),
    .priv_class     = &hevc_decoder_class,
    .init           = hevc_decode_init,
    .close          = hevc_decode_free,
}

```



```

.decode          = hevc_decode_frame,
.flush           = hevc_decode_flush,
.update_thread_context = hevc_update_thread_context,
.init_thread_copy = hevc_init_thread_copy,
.capabilities     = AV_CODEC_CAP_DR1 | AV_CODEC_CAP_DELAY |
    AV_CODEC_CAP_SLICE_THREADS | AV_CODEC_CAP_FRAME_THREADS,
.profiles         = NULL_IF_CONFIG_SMALL(ff_hevc_profiles),
};

```

从定义可以看出，`decode` 指向 `hevc_decode_frame` 函数，在 `hevc.c` 中我们找到了 `hevc_decode_frame` 函数，其代码如下：

```

static int hevc_decode_frame(AVCodecContext *avctx, void *data, int *got_
output, AVPacket *avpkt)
{
    int ret;
    HEVCContext *s = avctx->priv_data;
    if (!avpkt->size) {
        ret = ff_hevc_output_frame(s, data, 1);
        if (ret < 0)
            return ret;
        *got_output = ret;
        return 0;
    }
    s->ref = NULL;
    ret = decode_nal_units(s, avpkt->data, avpkt->size);
    if (ret < 0)
        return ret;
    if (avctx->hwaccel) {
        if (s->ref && (ret = avctx->hwaccel->end_frame(avctx)) < 0) {
            av_log(avctx, AV_LOG_ERROR,
                "hardware accelerator failed to decode picture\n");
            ff_hevc_unref_frame(s, s->ref, ~0);
            return ret;
        }
    }
    } else {
        /*校验 SEI (SEI 即补充增强信息 (Supplemental Enhancement Information)，属于码
        流范畴，它提供了向视频码流中加入额外信息的方法，是 H.264/H.265 这些视频压缩标准的特性之
        一，它的作用是在音视频内部传递消息) */
        if (avctx->err_recognition & AV_EF_CRCHECK && s->is_decoded &&
            s->is_md5) {
            ret = verify_md5(s, s->ref->frame);
            if (ret < 0 && avctx->err_recognition & AV_EF_EXPLODE) {
                ff_hevc_unref_frame(s, s->ref, ~0);
                return ret;
            }
        }
    }
}

```


Android 音视频开发

```

    }
}
s->is_md5 = 0;
if (s->is_decoded) {
    av_log(avctx, AV_LOG_DEBUG, "Decoded frame with POC %d.\n", s->
poc);
    s->is_decoded = 0;
}
if (s->output_frame->buf[0]) {
    av_frame_move_ref(data, s->output_frame);
    *got_output = 1;
}
return avpkt->size;
}

```

下面调用 `decode_nal_units` 函数完成 H.265 (HEVC) 的解码:

```

static int decode_nal_units(HEVCContext *s, const uint8_t *buf, int length)
{
    int i, ret = 0;
    s->ref = NULL;
    s->last_eos = s->eos;
    s->eos = 0;
    ret = ff_h2645_packet_split(&s->pkt, buf, length, s->avctx, s->is_
nalff, s->nal_length_size, s->avctx->codec_id, 1);
    if (ret < 0) {
        av_log(s->avctx, AV_LOG_ERROR,
            "Error splitting the input into NAL units.\n");
        return ret;
    }
    for (i = 0; i < s->pkt.nb_nals; i++) {
        if (s->pkt.nals[i].type == NAL_EOB_NUT ||
            s->pkt.nals[i].type == NAL_EOS_NUT)
            s->eos = 1;
    }
    /*解码 NAL 单元*/
    for (i = 0; i < s->pkt.nb_nals; i++) {
        ret = decode_nal_unit(s, &s->pkt.nals[i]);
        if (ret < 0) {
            av_log(s->avctx, AV_LOG_WARNING,
                "Error parsing NAL unit #%d. \n", i);
            goto fail;
        }
    }

fail:
    if (s->ref && s->threads_type == FF_THREAD_FRAME)

```

```

        ff_thread_report_progress(&s->ref->tf, INT_MAX, 0);
    return ret;
}

```

这个函数主要包含了如下两个步骤。

(1) `ff_h2645_packet_split()`，位于 `h2645_parser.c` 中，主要是对输入的音视频压缩数据包分离出 NAL 单元（下文简称 NALU），这样我们就知道解码后有多少个 slice。

(2) `decode_nal_unit`，我们跟踪进去看看它的功能，代码如下：

```

static int decode_nal_unit(HEVCContext *s, const H2645NAL *nal)
{
    HEVCLocalContext *lc = s->HEVCLc;
    GetBitContext *gb     = &lc->gb;
    int ctb_addr_ts, ret;

    *gb                = nal->gb;
    s->nal_unit_type = nal->type;
    s->temporal_id   = nal->temporal_id;
    switch (s->nal_unit_type) {
    case NAL_VPS:
        ret = ff_hevc_decode_nal_vps(gb, s->avctx, &s->ps);
        if (ret < 0)
            goto fail;
        break;
    case NAL_SPS:
        ret = ff_hevc_decode_nal_sps(gb, s->avctx, &s->ps,
                                     s->apply_defdispwin);
        if (ret < 0)
            goto fail;
        break;
    case NAL_PPS:
        ret = ff_hevc_decode_nal_pps(gb, s->avctx, &s->ps);
        if (ret < 0)
            goto fail;
        break;
    case NAL_SEI_PREFIX:
    case NAL_SEI_SUFFIX:
        ret = ff_hevc_decode_nal_sei(s);
        if (ret < 0)
            goto fail;
        break;
    case NAL_TRAIL_R:
    case NAL_TRAIL_N:
    case NAL_TSA_N:
    case NAL_TSA_R:

```



```

    case NAL_STSA_N:
    case NAL_STSA_R:
    case NAL_BLA_W_LP:
    case NAL_BLA_W_RADL:
    case NAL_BLA_N_LP:
    case NAL_IDR_W_RADL:
    case NAL_IDR_N_LP:
    case NAL_CRA_NUT:
    case NAL_RADL_N:
    case NAL_RADL_R:
    case NAL_RASL_N:
    case NAL_RASL_R:
        ret = hls_slice_header(s);
        if (ret < 0)
            return ret;
        if (s->max_ra == INT_MAX) {
            if (s->nal_unit_type == NAL_CRA_NUT || IS_BLA(s)) {
                s->max_ra = s->poc;
            } else {
                if (IS_IDR(s))
                    s->max_ra = INT_MIN;
            }
        }
        if ((s->nal_unit_type == NAL_RASL_R || s->nal_unit_type == NAL_
RASL_N) && s->poc <= s->max_ra) {
            s->is_decoded = 0;
            break;
        } else {
            if (s->nal_unit_type == NAL_RASL_R && s->poc > s->max_ra)
                s->max_ra = INT_MIN;
        }
        if (s->sh.first_slice_in_pic_flag) {
            ret = hevc_frame_start(s);
            if (ret < 0)
                return ret;
        } else if (!s->ref) {
            av_log(s->avctx, AV_LOG_ERROR, "First slice in a frame
            missing.\n");
            goto fail;
        }
        if (s->nal_unit_type != s->first_nal_type) {
            av_log(s->avctx, AV_LOG_ERROR,
                "Non-matching NAL types of the VCL NALUs: %d %d\n",
                s->first_nal_type, s->nal_unit_type);
            return AERROR_INVALIDDATA;
        }

```



```

        if (!s->sh.dependent_slice_segment_flag &&
            s->sh.slice_type != I_SLICE) {
            ret = ff_hevc_slice_rpl(s);
            if (ret < 0) {
                av_log(s->avctx, AV_LOG_WARNING,
                    "Error constructing the reference lists for the
                     current slice.\n");
                goto fail;
            }
        }
    }
    if (s->sh.first_slice_in_pic_flag && s->avctx->hwaccel) {
        ret = s->avctx->hwaccel->start_frame(s->avctx, NULL, 0);
        if (ret < 0)
            goto fail;
    }
    if (s->avctx->hwaccel) {
        ret = s->avctx->hwaccel->decode_slice(s->avctx, nal-> raw_
data, nal->raw_size);
        if (ret < 0)
            goto fail;
    } else {
        if (s->threads_number > 1 && s->sh.num_entry_point_offsets > 0)
            ctb_addr_ts = hls_slice_data_wpp(s, nal);
        else
            ctb_addr_ts = hls_slice_data(s);
        if (ctb_addr_ts >= (s->ps.sps->ctb_width * s->ps.sps-> ctb_
height)) {
            s->is_decoded = 1;
        }
        if (ctb_addr_ts < 0) {
            ret = ctb_addr_ts;
            goto fail;
        }
    }
    break;
case NAL_EOS_NUT:
case NAL_EOB_NUT:
    s->seq_decode = (s->seq_decode + 1) & 0xff;
    s->max_ra      = INT_MAX;
    break;
case NAL_AUD:
case NAL_FD_NUT:
    break;
default:
    av_log(s->avctx, AV_LOG_INFO,
        "Skipping NAL unit %d\n", s->nal_unit_type);

```

```

    }
    return 0;
fail:
    if (s->avctx->err_recognition & AV_EF_EXPLODE)
        return ret;
    return 0;
}

```

解码 NALU，可以看到就是根据 HEVC 的码流结构（在后面的章节中会介绍 HEVC 码流结构），按照 VPS、SPS、PPS（这些都是 NALU 的类型，H.264 没有 VPS），直到 EOS（End Of Stream），在判断出每个类型后，调用对应的函数解码 NALU，如 VPS 类型的 NALU 就调用 `ff_hevc_decode_nal_vps` 函数，SPS 类型的 NALU 就调用 `ff_hevc_decode_nal_sps` 函数。

8.3 FFmpeg 案例（代码实现）

前面我们对 FFmpeg API 的主要结构体以及关键函数做了一些介绍，本节主要通过一些 FFmpeg 实战案例，来加深读者对 FFmpeg 的认识。

8.3.1 利用 FFmpeg 转换格式

背景：需要将 MP4 格式的视频转换成 AVI 封装格式。案例运行图如图 8-4 所示

```

Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'C:\test\test.mp4':
  Metadata:
    major_brand      : mp42
    minor_version    : 0
    compatible_brands: isommp42
    creation_time    : 2016-12-13 09:46:18
  Duration: 00:00:09.62, start: 0.000000, bitrate: 1793 kb/s
  Stream #0:0(ceng): Video: h264 (High) (avc1 / 0x31637661), yuv420p, 1920x1080, 1790 kb/s, SAR 1:1 DAR 16:9, 25.78 fps, 90k tbr, 90k tbn, 180k tbc (default)
  Metadata:
    creation_time    : 2016-12-13 09:46:18
    handler_name     : VideoHandle
Output #0, avi, to 'C:\test\test.avi':
  Stream #0:0: Video: h264 (High), yuv420p, 1920x1080, q-2-31, 1790 kb/s, 18 tbc
[00:00:00.000000] Using AVStream.codec.time_base as a timebase hint to the muxer is deprecated. Set AVStream.time_base instead.
Write      0 frames to output file
Write      1 frames to output file
Write      2 frames to output file
Write      3 frames to output file
Write      4 frames to output file
Write      5 frames to output file
Write      6 frames to output file
Write      7 frames to output file
Write      8 frames to output file
Write      9 frames to output file

```

图 8-4 MP4 转 AVI 格式案例运行图

生成的文件如图 8-5 所示。



图 8-5 MP4 转 AVI 后生成的文件

生成完成后看看是否真正变成 AVI 封装格式，使用 `ffmpeg` 命令查看视频文件信息，如图 8-6 所示。

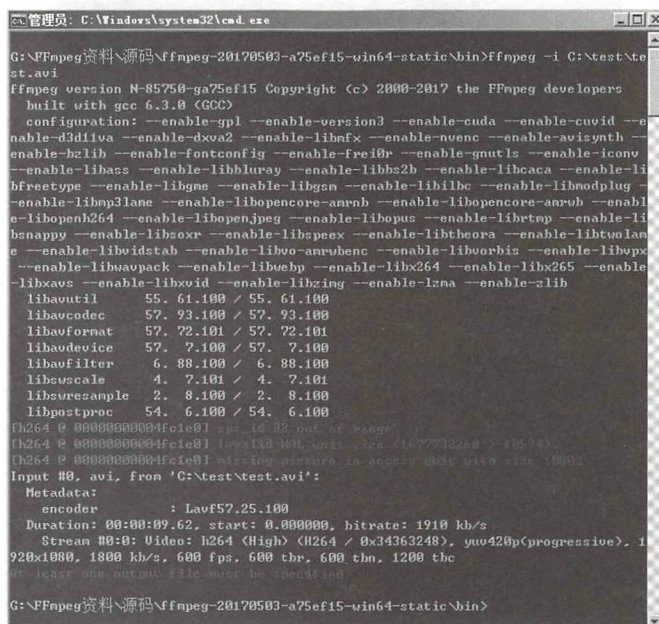


图 8-6 通过 ffmpeg 命令查看视频文件信息

可以看到 `input #0` 后就是 `avi`。这时有人会提出疑问，如果只改后缀名，能不能真正地改变封装格式。下面不妨做一个实验，把 `test.mp4` 改成 `test2.avi`，看看结果，如图 8-7 所示。

可以看到虽然人为地把 `mp4` 后缀名改成了 `avi`，但通过 `ffmpeg` 命令查看，发现内部封装容器依旧是 `mp4`，码率等很多参数都没有改变。


```

C:\Windows\system32\cmd.exe
G:\FFmpeg\资料\源码\ffmpeg-20170503-a75ef15-win64-static\bin>ffmpeg -i C:\test\test2.avi
ffmpeg version N-85750-ga75ef15 Copyright (c) 2000-2017 the FFmpeg developers
  built with gcc 6.3.0 (GCC)
  configuration: --enable-gpl --enable-version3 --enable-cuda --enable-cuvid --enable-d3d11va --enable-dxva2 --enable-libaom --enable-libbrotli --enable-libfreetype --enable-libgsm --enable-liblame --enable-liblibvpx --enable-libmodplug --enable-libmp3lame --enable-libopenjpeg --enable-libopenm264 --enable-libopenh264 --enable-libopus --enable-librtmp --enable-libsnappy --enable-libsoxr --enable-libspeex --enable-libtheora --enable-libtesseract --enable-libvidstab --enable-libvo-amrwbenc --enable-libvorbis --enable-libvpx --enable-libx264 --enable-libx265 --enable-libxavs --enable-libxvid --enable-libzimg --enable-lzma --enable-zlib
  libavutil 55. 61.100 / 55. 61.100
  libavcodec 57. 92.100 / 57. 92.100
  libavformat 57. 72.101 / 57. 72.101
  libavdevice 57.  7.100 / 57.  7.100
  libavfilter 6. 88.100 / 6. 88.100
  libswscale 4.  7.101 / 4.  7.101
  libswresample 2.  8.100 / 2.  8.100
  libpostproc 54.  6.100 / 54.  6.100
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'C:\test\test2.avi':
  Metadata:
    major_brand      : mp42
    minor_version    : 0
    compatible_brands: isommp42
    creation_time   : 2016-12-13T09:46:18.000000Z
  Duration: 00:00:09.62, start: 0.000000, bitrate: 1793 kb/s
  Stream #0:0(eng): Video: h264 (High) (avc1 / 0x31637661), yuv420p, 1920x1080, 1790 kb/s, SAR 1:1 DAR 16:9, 25.78 fps, 90k tbr, 90k tbn, 180k tbc (default)
  Metadata:
    creation_time   : 2016-12-13T09:46:18.000000Z
    handler_name    : VideoHandler
  
```

图 8-7 通过 ffmpeg 命令查看修改文件后缀名后的信息

FFmpeg 转 MP4 格式为 AVI 格式源码如下。

工程头文件:

```

extern "C"
{
    #include "libavutil/opt.h"
    #include "libavutil/channel_layout.h"
    #include "libavutil/common.h"
    #include "libavutil/imgutils.h"
    #include "libavutil/mathematics.h"
    #include "libavutil/samplefmt.h"
    #include "libavutil/time.h"
    #include "libavutil/fifo.h"
    #include "libavcodec/avcodec.h"
    #include "libavformat/avformat.h"
    #include "libavformat/avio.h"
    #include "libavfilter/avfiltergraph.h"
    #include "libavfilter/avfilter.h"
    #include "libavfilter/buffersink.h"
    #include "libavfilter/buffersrc.h"
    #include "libswscale/swscale.h"
    #include "libswresample/swresample.h"
}

```

主干文件的实现代码:

```
int _tmain(int argc, _TCHAR* argv[])
{
    AVOutputFormat *ofmt = NULL;
    AVBitStreamFilterContext *vbsf = NULL;
    //定义输入、输出 AVFormatContext
    AVFormatContext *ifmt_ctx = NULL, *ofmt_ctx = NULL;
    AVPacket pkt;
    const char *in_filename, *out_filename;
    int ret, i;
    int frame_index = 0;
    in_filename = "C:\\test\\test.mp4";//Input file URL
    out_filename = "C:\\test\\test.avi";//Output file URL

    av_register_all();
    //输入
    if ((ret = avformat_open_input(&ifmt_ctx, in_filename, 0, 0)) < 0)
    {
        //打开媒体文件
        printf("Could not open input file.");
        goto end;
    }
    if ((ret = avformat_find_stream_info(ifmt_ctx, 0)) < 0) {
        //获取视频信息
        printf("Failed to retrieve input stream information");
        goto end;
    }
    //MP4 中使用的是 H.264 编码, 而 H.264 编码有两种封装模式
    //一种是 annexb 模式, 它是传统模式, 有 startcode, SPS 和 PPS 在 Element Stream
    //中; 另一种是 mp4 模式, 一般 MP4、MKV、AVI 都没有 startcode, SPS 和 PPS 以及其
    //他信息被封装在容器中
    //每一帧前面是这一帧的长度值, 很多解码器只支持 annexb 模式, 因此需要对 MP4 模式做
    //转换在 FFmpeg 中用 h264_mp4toannexb_filter 可以进行模式转换; 使用命令
    //- bsf h264_mp4toannexb 就可实现转换

    vbsf = av_bitstream_filter_init("h264_mp4toannexb");

    av_dump_format(ifmt_ctx, 0, in_filename, 0);
    //初始化输出视频码流的 AVFormatContext
    avformat_alloc_output_context2(&ofmt_ctx, NULL, NULL, out_filename);
    if (!ofmt_ctx) {
        printf("Could not create output context\n");
        ret = AVERROR_UNKNOWN;
        goto end;
    }
    ofmt = ofmt_ctx->oformat;
```



```

    for (i = 0; i < ifmt_ctx->nb_streams; i++) {
        //通过输入的 AVStream 创建输出的 AVStream
        AVStream *in_stream = ifmt_ctx->streams[i];
        AVStream *out_stream = avformat_new_stream(ofmt_ctx, in_stream->
codec->codec); //初始化 AVStream
        if (!out_stream) {
            printf("Failed allocating output stream\n");
            ret = AERROR_UNKNOWN;
            goto end;
        }
        //复制 AVCodecContext 的设置属性
        if (avcodec_copy_context(out_stream->codec, in_stream->codec) < 0) {
            printf("Failed to copy context from input to output stream
codec context\n");
            goto end;
        }
        out_stream->codec->codec_tag = 0;
        if (ofmt_ctx->oformat->flags & AVFMT_GLOBALHEADER)
            out_stream->codec->flags |= CODEC_FLAG_GLOBAL_HEADER;
    }
    //输出信息
    av_dump_format(ofmt_ctx, 0, out_filename, 1);
    //打开输出文件
    if (!(ofmt->flags & AVFMT_NOFILE)) {
        ret = avio_open(&ofmt_ctx->pb, out_filename, AVIO_FLAG_WRITE);
        //打开输出文件
        if (ret < 0) {
            printf("Could not open output file '%s'", out_filename);
            goto end;
        }
    }
    //写文件头
    if (avformat_write_header(ofmt_ctx, NULL) < 0) {
        printf("Error occurred when opening output file\n");
        goto end;
    }

    while (1) {
        AVStream *in_stream, *out_stream;
        //得到一个 AVPacket
        ret = av_read_frame(ifmt_ctx, &pkt);
        if (ret < 0)
            break;
        in_stream = ifmt_ctx->streams[pkt.stream_index];
    }

```



```

        out_stream = ofmt_ctx->streams[pkt.stream_index];

        //转换 PTS/DTS
        pkt.pts = av_rescale_q_rnd(pkt.pts, in_stream->time_base, out_stream->time_base, (AVRounding) (AV_ROUND_NEAR_INF | AV_ROUND_PASS_MINMAX));
        pkt.dts = av_rescale_q_rnd(pkt.dts, in_stream->time_base, out_stream->time_base, (AVRounding) (AV_ROUND_NEAR_INF | AV_ROUND_PASS_MINMAX));
        pkt.duration = av_rescale_q(pkt.duration, in_stream->time_base, out_stream->time_base);
        pkt.pos = -1;

        if (pkt.stream_index == 0) {
            AVPacket fpkt = pkt;
            int a = av_bitstream_filter_filter(vbsf,
                out_stream->codec, NULL, &fpkt.data, &fpkt.size,
                pkt.data, pkt.size, pkt.flags & AV_PKT_FLAG_KEY);
            pkt.data = fpkt.data;
            pkt.size = fpkt.size;
        }
        //写 AVPacket
        if (av_write_frame(ofmt_ctx, &pkt) < 0) {
            //将 AVPacket (存储音视频压缩码流数据) 写入文件
            printf("Error muxing packet\n");
            break;
        }
        printf("Write %8d frames to output file\n", frame_index);
        av_packet_unref(&pkt);
        frame_index++;
    }
    //写文件尾
    av_write_trailer(ofmt_ctx);
end:
    avformat_close_input(&ifmt_ctx);
    /*关闭输出*/
    if (ofmt_ctx && !(ofmt->flags & AVFMT_NOFILE))
        avio_close(ofmt_ctx->pb);
    avformat_free_context(ofmt_ctx);
    system("pause");
    return 0;
}

```

8.3.2 在实时流中抓取图像

背景：通过直播流截取某一视频帧图像，如用作缩略图的场景。案例效果如图 8-8 所示。

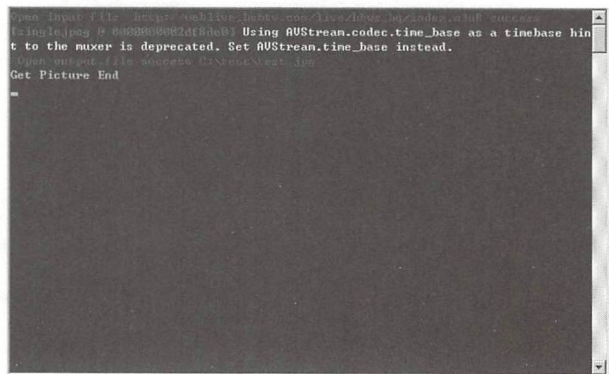


图 8-8 FFmpeg 通过直播流截取某一视频帧图像的案例效果

看对应的 C 盘的 test 目录，可查看案例运行后生成的文件，如图 8-9 所示。



图 8-9 案例运行后生成的文件

打开该图像，这是河北卫视直播流中的某一时刻图像，如图 8-10 所示。



图 8-10 抓取的河北卫视直播流中的某一时刻图像

从实时流中抓取图像源码如下。

工程头文件：

```
extern "C"
```

```

{
#include "libavutil/opt.h"
#include "libavutil/channel_layout.h"
#include "libavutil/common.h"
#include "libavutil/imgutils.h"
#include "libavutil/mathematics.h"
#include "libavutil/samplefmt.h"
#include "libavutil/time.h"
#include "libavutil/fifo.h"
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libavformat/avio.h"
#include "libavfilter/avfiltergraph.h"
#include "libavfilter/avfilter.h"
#include "libavfilter/buffersink.h"
#include "libavfilter/buffersrc.h"
#include "libswscale/swscale.h"
#include "libswresample/swresample.h"
}

```

主干文件的实现代码:

```

#include "stdafx.h"
#include <string>
#include <memory>
#include <thread>
#include <iostream>
#include <iostream>
using namespace std;
#include "ffmpegheader.h"

AVFormatContext *inputContext = nullptr;
AVFormatContext * outputContext;
int64_t lastReadPcktTime ;

static int interrupt_cb(void *ctx)
{
    int timeout = 3;
    if(av_gettime() - lastReadPcktTime > timeout *1000 *1000)
    {
        return -1;
    }
    return 0;
}

```



```

int OpenInput(string inputUrl)
{
    inputContext = avformat_alloc_context();
    lastReadPacktTime = av_gettime();
    inputContext->interrupt_callback.callback = interrupt_cb;
    int ret = avformat_open_input(&inputContext, inputUrl.c_str(), nullptr,
    nullptr);
    if(ret < 0)
    {
        av_log(NULL, AV_LOG_ERROR, "Input file open input failed\n");
        return ret;
    }
    ret = avformat_find_stream_info(inputContext, nullptr);
    if(ret < 0)
    {
        av_log(NULL, AV_LOG_ERROR, "Find input file stream inform failed\n");
    }
    else
    {
        av_log(NULL, AV_LOG_FATAL, "Open input file %s success\n", inputUrl.
c_str());
    }
    return ret;
}

shared_ptr<AVPacket> ReadPacketFromSource()
{
    shared_ptr<AVPacket> packet(static_cast<AVPacket*> (av_malloc(sizeof
(AVPacket))), [&](AVPacket *p) { av_packet_free(&p); av_freep(&p);});
    av_init_packet(packet.get());
    lastReadPacktTime = av_gettime();
    int ret = av_read_frame(inputContext, packet.get());
    if(ret >= 0)
    {
        return packet;
    }
    else
    {
        return nullptr;
    }
}

int OpenOutput(string outUrl)

```

```

{
    int ret = avformat_alloc_output_context2(&outputContext, nullptr,
"singlejpeg", outUrl.c_str());
    if(ret < 0)
    {
        av_log(NULL, AV_LOG_ERROR, "open output context failed\n");
        goto Error;
    }

    ret = avio_open2(&outputContext->pb, outUrl.c_str(), AVIO_FLAG_WRITE,
nullptr, nullptr);
    if(ret < 0)
    {
        av_log(NULL, AV_LOG_ERROR, "open avio failed");
        goto Error;
    }
    for(int i = 0; i < inputContext->nb_streams; i++)
    {
        if(inputContext->streams[i]->codec->codec_type == AVMediaType::
AVMEDIA_TYPE_AUDIO)
        {
            continue;
        }
        AVStream * stream = avformat_new_stream(outputContext, inputContext ->
streams[i]->codec->codec);
        ret = avcodec_copy_context(stream->codec, inputContext-> streams[i]
->codec);
        if(ret < 0)
        {
            av_log(NULL, AV_LOG_ERROR, "copy codec context failed");
            goto Error;
        }
    }

    ret = avformat_write_header(outputContext, nullptr);
    if(ret < 0)
    {
        av_log(NULL, AV_LOG_ERROR, "format write header failed");
        goto Error;
    }

    av_log(NULL, AV_LOG_FATAL, " Open output file success %s\n", outUrl.
c_str());
}

```



```

        return ret ;
Error:
    if(outputContext)
    {
        for(int i = 0; i < outputContext->nb_streams; i++)
        {
            avcodec_close(outputContext->streams[i]->codec);
        }
        avformat_close_input(&outputContext);
    }
    return ret ;
}

void Init()
{
    av_register_all();
    avfilter_register_all();
    avformat_network_init();
    av_log_set_level(AV_LOG_WARNING);
}

void CloseInput()
{
    if(inputContext != nullptr)
    {
        avformat_close_input(&inputContext);
    }
}

void CloseOutput()
{
    if(outputContext != nullptr)
    {
        int ret = av_write_trailer(outputContext);
        for(int i = 0 ; i < outputContext->nb_streams; i++)
        {
            AVCodecContext *codecContext = outputContext->streams[i]-> codec;
            avcodec_close(codecContext);
        }
        avformat_close_input(&outputContext);
    }
}

```



```

int WritePacket(shared_ptr<AVPacket> packet)
{
    auto inputStream = inputContext->streams[packet->stream_index];
    auto outputStream = outputContext->streams[packet->stream_index];
    return av_interleaved_write_frame(outputContext, packet.get());
}

int InitDecodeContext(AVStream *inputStream)
{
    auto codecId = inputStream->codec->codec_id;
    auto codec = avcodec_find_decoder(codecId);
    if (!codec)
    {
        return -1;
    }

    int ret = avcodec_open2(inputStream->codec, codec, NULL);
    return ret;
}

int initEncoderCodec(AVStream* inputStream, AVCodecContext **encodeContext)
{
    AVCodec * picCodec;

    picCodec = avcodec_find_encoder(AV_CODEC_ID_MJPEG);
    (*encodeContext) = avcodec_alloc_context3(picCodec);

    (*encodeContext)->codec_id = picCodec->id;
    (*encodeContext)->time_base.num = inputStream->codec->time_base.num;
    (*encodeContext)->time_base.den = inputStream->codec->time_base.den;
    (*encodeContext)->pix_fmt = *picCodec->pix_fmts;
    (*encodeContext)->width = inputStream->codec->width;
    (*encodeContext)->height = inputStream->codec->height;
    int ret = avcodec_open2((*encodeContext), picCodec, nullptr);
    if (ret < 0)
    {
        std::cout<<"open video codec failed"<<endl;
        return ret;
    }

    return 1;
}

bool Decode(AVStream* inputStream, AVPacket* packet, AVFrame *frame)

```

```

{
    int gotFrame = 0;
    auto hr = avcodec_decode_video2(inputStream->codec, frame, &gotFrame,
packet);
    if (hr >= 0 && gotFrame != 0)
    {
        return true;
    }
    return false;
}

std::shared_ptr<AVPacket> Encode(AVCodecContext *encodeContext, AVFrame *
frame)
{
    int gotOutput = 0;
    std::shared_ptr<AVPacket> pkt(static_cast<AVPacket*> (av_malloc(sizeof
(AVPacket))), [&](AVPacket *p) { av_packet_free(&p); av_freep(&p); });
    av_init_packet(pkt.get());
    pkt->data = NULL;
    pkt->size = 0;
    int ret = avcodec_encode_video2(encodeContext, pkt.get(), frame,
&gotOutput);
    if (ret >= 0 && gotOutput)
    {
        return pkt;
    }
    else
    {
        return nullptr;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    Init();
    int ret = OpenInput("http://weblive.hebtv.com/live/hbws_bq/index. m3u8");
    if(ret >= 0)
    {
        ret = OpenOutput("C:\\test\\test.jpg");
    }
    if(ret <0) goto Error;

    AVCodecContext *encodeContext = nullptr;

```



```

InitDecodeContext(inputContext->streams[0]);
AVFrame *videoFrame = av_frame_alloc();
initEncoderCodec(inputContext->streams[0], &encodeContext);

while(true)
{
    auto packet = ReadPacketFromSource();
    if(packet && packet->stream_index == 0)
    {
        if(Decode(inputContext->streams[0], packet.get(), videoFrame))
        {
            auto packetEncode = Encode(encodeContext, videoFrame);
            if(packetEncode)
            {
                ret = WritePacket(packetEncode);
                if(ret >= 0)
                {
                    break;
                }
            }
        }
    }
    cout << "Get Picture End " << endl;
    av_frame_free(&videoFrame);
    avcodec_close(encodeContext);
    Error:
    CloseInput();
    CloseOutput();

    while(true)
    {
        this_thread::sleep_for(chrono::seconds(100));
    }
    return 0;
}

```

8.3.3 在视频中加入水印

我们依旧使用 test.mp4 作为测试文件，把 logo.png 嵌入 test.mp4 文件中，生成 test_watermark.mp4 文件。我们的所有文件如图 8-11 所示。



名称	日期	类型	大小	长度
logo.png	2017/10/28 11:57	图片文件 (.png)	15 KB	
test.avi	2017/10/28 10:23	媒体文件 (.avi)	2,244 KB	00:00:09
test.jpg	2017/10/28 10:06	图片文件 (.jpg)	28 KB	
test.mp4	2016/12/13 19:46	媒体文件 (.mp4)	2,107 KB	00:00:09
test_watermark.mp4	2017/10/28 11:59	媒体文件 (.mp4)	10,251 KB	
test2.avi	2016/12/13 19:46	媒体文件 (.avi)	2,107 KB	
乞丐营销策略.mp4	2017/2/25 11:59	媒体文件 (.mp4)	16,134 KB	00:05:51

图 8-11 我们的所有文件

使用 ffplay 播放 test_watermark.mp4 文件，水印效果如图 8-12 所示。



图 8-12 使用 ffplay 播放加水印后的视频

在视频中加入水印的工程源码如下（头文件和其他案例一样，这里省略）：

```
#include "stdafx.h"
#include "ffmpegheader.h"
#include <string>
#include <iostream>
#include <thread>
#include <memory>
#include <iostream>
#include <fstream>
#include <Winsock2.h>
```

```

#include <condition_variable>
#include <concurrent_queue.h>

using namespace std;

AVFormatContext* context[2];
AVFormatContext* outputContext;
int64_t lastPts = 0;
int64_t lastDts = 0;
int64_t lastFrameRealtime = 0;

int64_t firstPts = AV_NOPTS_VALUE;
int64_t startTime = 0;

AVCodecContext* outPutEncContext = NULL;
AVCodecContext* decoderContext[2];

#define SrcWidth 1920
#define SrcHeight 1080
#define DstWidth 640
#define DstHeight 480

struct SwsContext* pSwsContext;

AVFilterInOut* inputs;
AVFilterInOut* outputs;
AVFilterGraph* filter_graph = nullptr;

AVFilterContext* inputFilterContext[2];
AVFilterContext* outputFilterContext = nullptr;
const char *filter_descr = "overlay=100:100";

int interrupt_cb(void *ctx)
{
    return 0;
}

void Init()
{
    av_register_all();
    avfilter_register_all();
    avformat_network_init();
    avdevice_register_all();
    av_log_set_level(AV_LOG_ERROR);
}

```



```

}

int OpenInput(char *fileName,int inputIndex)
{
    context[inputIndex] = avformat_alloc_context();
    context[inputIndex]->interrupt_callback.callback = interrupt_cb;
    AVDictionary *format_opts = nullptr;

    int ret = avformat_open_input(&context[inputIndex], fileName, nullptr,
&format_opts);
    if(ret < 0)
    {
        return ret;
    }
    ret = avformat_find_stream_info(context[inputIndex],nullptr);
    av_dump_format(context[inputIndex], 0, fileName, 0);
    if(ret >= 0)
    {
        std::cout <<"open input stream successfully" << endl;
    }
    return ret;
}

shared_ptr<AVPacket> ReadPacketFromSource(int inputIndex)
{
    std::shared_ptr<AVPacket> packet(static_cast<AVPacket*>(av_malloc (sizeof
(AVPacket))), [&](AVPacket *p) { av_packet_free(&p); av_freep(&p); });
    av_init_packet(packet.get());
    int ret = av_read_frame(context[inputIndex], packet.get());
    if(ret >= 0)
    {
        return packet;
    }
    else
    {
        return nullptr;
    }
}

int OpenOutput(char *fileName,int inputIndex)
{
    int ret = 0;
    ret = avformat_alloc_output_context2(&outputContext, nullptr, "mpegts",
fileName);

```




```
if(ret < 0)
{
    goto Error;
}
ret = avio_open2(&outputContext->pb, fileName, AVIO_FLAG_READ_WRITE,
nullptr, nullptr);
if(ret < 0)
{
    goto Error;
}

for(int i = 0; i < context[inputIndex]->nb_streams; i++)
{
    AVStream * stream = avformat_new_stream(outputContext, outPutEncContext
->codec);
    stream->codec = outPutEncContext;
    if(ret < 0)
    {
        goto Error;
    }
}
av_dump_format(outputContext, 0, fileName, 1);
ret = avformat_write_header(outputContext, nullptr);
if(ret < 0)
{
    goto Error;
}
if(ret >= 0)
    cout <<"open output stream successfully" << endl;
return ret ;

Error:
if(outputContext)
{
    avformat_close_input(&outputContext);
}
return ret ;
}

void CloseInput(int inputIndex)
{
    if(context != nullptr)
    {
        avformat_close_input(&context[inputIndex]);
    }
}
```



```
}

void CloseOutput()
{
    if(outputContext != nullptr)
    {
        for(int i = 0 ; i < outputContext->nb_streams; i++)
        {
            AVCodecContext *codecContext = outputContext->streams[i]-> codec;
            avcodec_close(codecContext);
        }
        avformat_close_input(&outputContext);
    }
}

int InitEncoderCodec( int iWidth, int iHeight,int inputIndex)
{
    AVCodec * pH264Codec = avcodec_find_encoder(AV_CODEC_ID_H264);
    if(NULL == pH264Codec)
    {
        printf("%s", "avcodec_find_encoder failed");
        return -1;
    }
    outPutEncContext = avcodec_alloc_context3(pH264Codec);
    outPutEncContext->gop_size = 30;
    outPutEncContext->has_b_frames = 0;
    outPutEncContext->max_b_frames = 0;
    outPutEncContext->codec_id = pH264Codec->id;
    outPutEncContext->time_base.num =context[inputIndex]->streams[0]->
codec->time_base.num;
    outPutEncContext->time_base.den = context[inputIndex]->streams[0]->
codec->time_base.den;
    outPutEncContext->pix_fmt          = *pH264Codec->pix_fmts;
    outPutEncContext->width            = iWidth;
    outPutEncContext->height           = iHeight;

    outPutEncContext->me_subpel_quality = 0;
    outPutEncContext->refs = 1;
    outPutEncContext->scenechange_threshold = 0;
    outPutEncContext->trellis = 0;
    AVDictionary *options = nullptr;
    outPutEncContext->flags |= AV_CODEC_FLAG_GLOBAL_HEADER;

    int ret = avcodec_open2(outPutEncContext, pH264Codec, &options);
```




```
if (ret < 0)
{
    printf("%s", "open codec failed");
    return ret;
}
return 1;
}

int InitDecodeCodec(AVCodecID codecId,int inputIndex)
{
    auto codec = avcodec_find_decoder(codecId);
    if(!codec)
    {
        return -1;
    }
    decoderContext[inputIndex] = context[inputIndex]->streams[0]->codec;
    if (!decoderContext) {
        fprintf(stderr, "Could not allocate video codec context\n");
        exit(1);
    }

    if (codec->capabilities & AV_CODEC_CAP_TRUNCATED)
        decoderContext[inputIndex]->flags |= AV_CODEC_FLAG_TRUNCATED;
    int ret = avcodec_open2(decoderContext[inputIndex], codec, NULL);
    return ret;
}

bool DecodeVideo(AVPacket* packet, AVFrame* frame,int inputIndex)
{
    int gotFrame = 0;
    auto hr = avcodec_decode_video2(decoderContext[inputIndex], frame,
&gotFrame, packet);
    if(hr >= 0 && gotFrame != 0)
    {
        frame->pts = packet->pts;
        return true;
    }
    return false;
}

int InitInputFilter(AVFilterInOut *input,const char *filterName, int
inputIndex)
{

```




```
char args[512];
memset(args,0, sizeof(args));
AVFilterContext *padFilterContext = input->filter_ctx;

    auto filter = avfilter_get_by_name("buffer");
    auto codecContext = context[inputIndex]->streams[0]->codec;

    sprintf_s(args, sizeof(args),
        "video_size=%dx%d:pix_fmt=%d:time_base=%d/%d:pixel_aspect=%d/%d",
        codecContext->width, codecContext->height, codecContext->pix_fmt,
        codecContext->time_base.num, codecContext->time_base.den /codecContext
        -> ticks_per_frame, codecContext->sample_aspect_ratio.num,
        codecContext-> sample_ aspect_ratio.den);

    int ret = avfilter_graph_create_filter(&inputFilterContext [inputIndex],
    filter, filterName, args, NULL, filter_graph);
    if(ret < 0) return ret;
    ret = avfilter_link(inputFilterContext[inputIndex],0, padFilterContext,
input->pad_idx);
    return ret;
}

int InitOutputFilter(AVFilterInOut *output,const char *filterName)
{
    AVFilterContext *padFilterContext = output->filter_ctx;
    auto filter = avfilter_get_by_name("buffersink");

    int ret = avfilter_graph_create_filter(&outputFilterContext,filter,
filterName, NULL, NULL, filter_graph);
    if(ret < 0) return ret;
    ret = avfilter_link(padFilterContext,output->pad_idx,
outputFilterContext,0);

    return ret;
}

void FreeInout()
{
    avfilter_inout_free(&inputs->next);
    avfilter_inout_free(&inputs);
    avfilter_inout_free(&outputs);
}
```



```
int _tmain(int argc, _TCHAR* argv[])
{
    string fileInput[2];
    fileInput[0] = "c:\\test\\test.mp4";
    fileInput[1] = "c:\\test\\logo.png";
    string fileOutput = "c:\\test\\test_watermark.mp4";
    std::thread decodeTask;
    Init();
    for(int i = 0; i < 2; i++)
    {
        if(OpenInput((char *)fileInput[i].c_str(),i) < 0)
        {
            cout << "Open file Input failed!" << endl;
            this_thread::sleep_for(chrono::seconds(10));
            return 0;
        }
    }
    for(int i = 0; i < 2; i++)
    {
        int ret = InitDecodeCodec(context[i]->streams[0]->codec->
codec_id, i);
        if(ret < 0)
        {
            cout << "InitDecodeCodec failed!" << endl;
            this_thread::sleep_for(chrono::seconds(10));
            return 0;
        }
    }

    int ret = InitEncoderCodec(decoderContext[0]->width, decoderContext[0]->
height,0);
    if(ret < 0)
    {
        cout << "open eccoder failed ret is " << ret<<endl;
        cout << "InitEncoderCodec failed!" << endl;
        this_thread::sleep_for(chrono::seconds(10));
        return 0;
    }

    filter_graph = avfilter_graph_alloc();
    if(!filter_graph)
    {
        cout <<"graph alloc failed"<<endl;
        goto End;
    }
}
```




```
}

avfilter_graph_parse2(filter_graph, filter_descr, &inputs, &outputs);
InitInputFilter(inputs, "MainFrame", 0);
InitInputFilter(inputs->next, "OverlayFrame", 1);
InitOutputFilter(outputs, "output");
FreeInout();

ret = avfilter_graph_config(filter_graph, NULL);
if(ret < 0)
{
    goto End;
}
if(OpenOutput((char *)fileOutput.c_str(), 0) < 0)
{
    cout << "Open file Output failed!" << endl;
    this_thread::sleep_for(chrono::seconds(10));
    return 0;
}

AVFrame *pSrcFrame[2];
AVFrame *inputFrame[2];
pSrcFrame[0] = av_frame_alloc();
pSrcFrame[1] = av_frame_alloc();
inputFrame[0] = av_frame_alloc();
inputFrame[1] = av_frame_alloc();
auto filterFrame = av_frame_alloc();
int got_output = 0;
int64_t timeRecord = 0;
int64_t firstPacketTime = 0;
int64_t outLastTime = av_gettime();
int64_t inLastTime = av_gettime();
int64_t videoCount = 0;
decodeTask.swap(thread([&]{
    bool ret = true;
    while(ret)
    {
        auto packet = ReadPacketFromSource(1);
        ret = DecodeVideo(packet.get(), pSrcFrame[1], 1);
        if(ret) break;
    }
})));
decodeTask.join();
```




```
while(true)
{
    outLastTime = av_gettime();
    auto packet = ReadPacketFromSource(0);
    if(packet)
    {
        if(DecodeVideo(packet.get(), pSrcFrame[0], 0))
        {
            av_frame_ref( inputFrame[0], pSrcFrame[0]);
            if (av_buffersrc_add_frame_flags(inputFilterContext[0],
inputFrame[0], AV_BUFFERSRC_FLAG_PUSH) >= 0)
            {
                pSrcFrame[1]->pts = pSrcFrame[0]->pts;
                //av_frame_ref( inputFrame[1], pSrcFrame[1]);
                if(av_buffersrc_add_frame_flags
(inputFilterContext[1], pSrcFrame[1], AV_BUFFERSRC_FLAG_PUSH) >= 0)
                {
                    ret = av_buffersink_get_frame_flags
(outputFilterContext, filterFrame, AV_BUFFERSINK_FLAG_NO_REQUEST);

                    if ( ret >= 0)
                    {
                        std::shared_ptr<AVPacket> pTmpPkt(static_
cast<AVPacket*>(av_malloc(sizeof(AVPacket))), [&](AVPacket *p)
{ av_packet_free(&p); av_freep(&p); });
                        av_init_packet(pTmpPkt.get());
                        pTmpPkt->data = NULL;
                        pTmpPkt->size = 0;
                        ret = avcodec_encode_video2(outPutEncContext,
pTmpPkt.get(), filterFrame, &got_output);
                        if (ret >= 0 && got_output)
                        {
                            int ret = av_write_frame(outputContext,
pTmpPkt.get());
                        }
                    }
                }
            }
            //this_thread::sleep_for(chrono::milliseconds(10));
            }
            av_frame_unref(filterFrame);
        }
    }
}
else break;
```



```
    }  
End:  
  
    CloseInput(0);  
    CloseInput(1);  
    CloseOutput();  
    std::cout <<"Transcode file end!" << endl;  
    this_thread::sleep_for(chrono::hours(10));  
    return 0;  
}
```

8.3.4 FFmpeg 音频解码

与前面不同的是，这是一个 Android Studio 项目，首先看看该项目中的 Java 代码：

```
package com.hejunlin.ffmpegaudio;  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.view.View;  
import android.widget.EditText;  
import android.widget.TextView;  
  
public class MainActivity extends AppCompatActivity {  
  
    private EditText mInput;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        mInput = (EditText) findViewById(R.id.et_input);  
        mInput.setText("http://ra01.sycdn.kuwo.cn/resource/n3/32/56/  
3260586875.mp3");  
        findViewById(R.id.bt_play).setOnClickListener(new View.  
OnClickListener() {  
            @Override  
            public void onClick(View view) {  
                NativePlayer.play(mInput.getText().toString().trim());  
            }  
        });  
        findViewById(R.id.bt_pause).setOnClickListener(new View.  
OnClickListener() {  
            @Override  
            public void onClick(View view) {  
                NativePlayer.stop();  
            }  
        });  
    }  
}
```




```
    }  
    });  
}  
}
```

NativePlayer 类的主要作用是调用 native 函数，代码如下：

```
package com.hejunlin.ffmpegaudio;  
  
public class NativePlayer {  
    static {  
        System.loadLibrary("NativePlayer");  
    }  
    public static native void play(String url);  
    public static native void stop();  
}
```

布局文件，即运行在手机 App 上 UI 展示的布局，其代码如下：

```
<?xml version="1.0" encoding="utf-8"?>  
<RelativeLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:id="@+id/activity_main"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    tools:context="com.hejunlin.ffmpegaudio.MainActivity">  
  
    <TextView  
        android:id="@+id/tv_input"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:padding="10dp"  
        android:layout_marginTop="30dp"  
        android:text="播放链接:"  
        android:textSize="20sp"/>  
  
    <EditText  
        android:id="@+id/et_input"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:layout_toRightOf="@id/tv_input"  
        android:padding="10dp"/>  
  
    <LinearLayout  
        android:layout_width="match_parent"
```




```
        android:layout_height="wrap_content"
        android:layout_below="@id/et_input"
        android:orientation="horizontal">

        <Button
            android:id="@+id/bt_play"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:layout_marginLeft="60dp"
            android:background="@drawable/button_shape"
            android:textColor="@color/white"
            android:text="播放" />

        <Button
            android:id="@+id/bt_pause"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:background="@drawable/button_shape"
            android:textColor="@color/white"
            android:layout_marginLeft="80dp"
            android:text="暂停" />
    </LinearLayout>
</RelativeLayout>
```

JNI 相关代码:

OpenSL_ES_Core.c

```
#include "OpenSL_ES_Core.h"
#include "FFmpegCore.h"
#include <assert.h>
#include <jni.h>
#include <string.h>

#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>

//调用 Native 的 AssetManager 类
#include <sys/types.h>
#include <android/asset_manager.h>
#include <android/asset_manager_jni.h>
#include "log.h"
```



```
//引擎接口
static SLObjectItf engineObject = NULL;
static SLEngineItf engineEngine;

//混合输出接口
static SLObjectItf outputMixObject = NULL;
static SLEnvironmentalReverbItf outputMixEnvironmentalReverb = NULL;

//播放器缓冲队列接口
static SLObjectItf bqPlayerObject = NULL;
static SLPlayItf bqPlayerPlay;
static SLAndroidSimpleBufferQueueItf bqPlayerBufferQueue;
static SLEffectSendItf bqPlayerEffectSend;
static SLMuteSoloItf bqPlayerMuteSolo;
static SLVolumeItf bqPlayerVolume;

//混合输出时的辅助效果，用于播放器缓冲队列
static const SLEnvironmentalReverbSettings reverbSettings =
    SL_I3DL2_ENVIRONMENT_PRESET_STONECORRIDOR;

static void *buffer;
static size_t bufferSize;

//每当缓冲区结束播放时，调用 bqPlayerCallback 函数
void bqPlayerCallback(SLAndroidSimpleBufferQueueItf bq, void *context)
{
    LOGD(">> buffere queue callback");
    assert(bq == bqPlayerBufferQueue);
    bufferSize = 0;
    //assert(NULL == context);
    getPCM(&buffer, &bufferSize);
    //寻找并且填充下一个缓冲区
    if (NULL != buffer && 0 != bufferSize) {
        SLresult result;
        //入队另一个缓冲区
        result = (*bqPlayerBufferQueue)->Enqueue(bqPlayerBufferQueue,
buffer, bufferSize);
        assert(SL_RESULT_SUCCESS == result);

        (void)result;
    }
}

void initOpenSLES()
{

```




```
LOGD(">> initOpenSLES...");
SLresult result;
//1.创建引擎
result = slCreateEngine(&engineObject, 0, NULL, 0, NULL, NULL);
LOGD(">> initOpenSLES... step 1, result = %d", result);
//2.实现引擎
result = (*engineObject)->Realize(engineObject, SL_BOOLEAN_FALSE);
LOGD(">> initOpenSLES...step 2, result = %d", result);
//3.得到引擎接口，用于创建其他对象
result = (*engineObject)->GetInterface(engineObject, SL_IID_ENGINE,
&engineEngine);
LOGD(">> initOpenSLES...step 3, result = %d", result);
//4.创建混合输出，将环境混响指定为非必需接口
const SLInterfaceID ids[1] = {SL_IID_ENVIRONMENTALREVERB};
const SLboolean req[1] = {SL_BOOLEAN_FALSE};
result = (*engineEngine)->CreateOutputMix(engineEngine,
&outputMixObject, 0, 0, 0);
LOGD(">> initOpenSLES...step 4, result = %d", result);
//5.实现混合输出
result = (*outputMixObject)->Realize(outputMixObject, SL_BOOLEAN_FALSE);
LOGD(">> initOpenSLES...step 5, result = %d", result);
//6.获取环境混响接口
//如果环境混响效果不可用，获取接口操作可能会失败，或者因为该特性不存在，CPU 负载过大，
//或者因为 MODIFY_AUDIO_SETTINGS 权限没有被授予，都会导致获取接口失败
result = (*outputMixObject)->GetInterface(outputMixObject, SL_IID_
ENVIRONMENTALREVERB, &outputMixEnvironmentalReverb);
if (SL_RESULT_SUCCESS == result) {
    result = (*outputMixEnvironmentalReverb)->SetEnvironmentalReverb-
Properties(outputMixEnvironmentalReverb, &reverbSettings);
    LOGD(">> initOpenSLES...step 6, result = %d", result);
}
}
//初始化缓冲队列
void initBufferQueue(int rate, int channel, int bitsPerSample)
{
    LOGD(">> initBufferQueue");
    SLresult result;
    //构建音频源
    SLDataLocator_AndroidSimpleBufferQueue loc_bufq = {SL_DATALOCATOR_
ANDROIDSIMPLEBUFFERQUEUE, 2};
    SLDataFormat_PCM format_pcm;
    format_pcm.formatType = SL_DATAFORMAT_PCM;
    format_pcm.numChannels = channel;
    format_pcm.samplesPerSec = rate * 1000;
```




```
format_pcm.bitsPerSample = bitsPerSample;
format_pcm.containerSize = 16;
if (channel == 2)
    format_pcm.channelMask = SL_SPEAKER_FRONT_LEFT | SL_SPEAKER_FRONT_
RIGHT;
else
    format_pcm.channelMask = SL_SPEAKER_FRONT_CENTER;
format_pcm.endianness = SL_BYTEORDER_LITTLEENDIAN;
SLDataSource audioSrc = {&loc_bufq, &format_pcm};
//构建音频后端
SLDataLocator_OutputMix loc_outmix = {SL_DATALOCATOR_OUTPUTMIX,
outputMixObject};
SLDataSink audioSnk = {&loc_outmix, NULL};
//创建音频播放器
const SLInterfaceID ids[3] = {SL_IID_BUFFERQUEUE, SL_IID_EFFECTSEND,
/*SL_IID_MUTESOLO,*/ SL_IID_VOLUME};
const SLboolean req[3] = {SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE,
/*SL_BOOLEAN_TRUE,*/ SL_BOOLEAN_TRUE};
result = (*engineEngine)->CreateAudioPlayer(engineEngine, &bqPlayerObject,
&audioSrc, &audioSnk, 3, ids, req);
assert(SL_RESULT_SUCCESS == result);
(void)result;
//实现音频播放器
result = (*bqPlayerObject)->Realize(bqPlayerObject, SL_BOOLEAN_FALSE);
assert(SL_RESULT_SUCCESS == result);
(void)result;
//获取播放接口
result = (*bqPlayerObject)->GetInterface(bqPlayerObject, SL_IID_PLAY,
&bqPlayerPlay);
assert(SL_RESULT_SUCCESS == result);
(void)result;
//获取缓冲队列接口
result = (*bqPlayerObject)->GetInterface(bqPlayerObject, SL_IID_
BUFFERQUEUE, &bqPlayerBufferQueue);
assert(SL_RESULT_SUCCESS == result);
(void)result;
//注册缓冲队列的回调函数
result = (*bqPlayerBufferQueue)->RegisterCallback(bqPlayerBuffer
Queue, bqPlayerCallback, NULL);
assert(SL_RESULT_SUCCESS == result);
(void)result;
//获取音效发送接口
result = (*bqPlayerObject)->GetInterface(bqPlayerObject, SL_IID_
EFFECTSEND, &bqPlayerEffectSend);
```



Android 音视频开发

```
    assert(SL_RESULT_SUCCESS == result);
    (void)result;
    //获取音量接口
    result = (*bqPlayerObject)->GetInterface(bqPlayerObject, SL_IID_
VOLUME, &bqPlayerVolume);
    assert(SL_RESULT_SUCCESS == result);
    (void)result;
    //设置播放器播放时的状态
    result = (*bqPlayerPlay)->SetPlayState(bqPlayerPlay, SL_PLAYSTATE_
PLAYING);
    assert(SL_RESULT_SUCCESS == result);
    (void)result;
}
//停止 Native 音频系统
void stop()
{
    //销毁缓冲队列音频播放器实例，置空所有关联接口
    if (bqPlayerObject != NULL) {
        (*bqPlayerObject)->Destroy(bqPlayerObject);
        bqPlayerObject = NULL;
        bqPlayerPlay = NULL;
        bqPlayerBufferQueue = NULL;
        bqPlayerEffectSend = NULL;
        bqPlayerMuteSolo = NULL;
        bqPlayerVolume = NULL;
    }
    //销毁混合输出实例，置空所有关联接口
    if (outputMixObject != NULL) {
        (*outputMixObject)->Destroy(outputMixObject);
        outputMixObject = NULL;
        outputMixEnvironmentalReverb = NULL;
    }
    //销毁引擎实例，置空所有关联接口
    if (engineObject != NULL) {
        (*engineObject)->Destroy(engineObject);
        engineObject = NULL;
        engineEngine = NULL;
    }
    //释放 FFmpeg 解码器
    releaseFFmpeg();
}
void play(char *url)
{
    int rate, channel;
```




```
LOGD("...get url=%s", url);
//1.初始化 FFmpeg 解码器
initFFmpeg(&rate, &channel, url);
//2.初始化 OpenSLES
initOpenSLES();
//3.初始化 BufferQueue
initBufferQueue(rate, channel, SL_PCMSAMPLEFORMAT_FIXED_16);
//4.启动音频播放
bqPlayerCallback(bqPlayerBufferQueue, NULL);
}
```

FFmpegCore.c 类的主要作用与 FFmpeg 解码播放相关，通过引用 OpenSLES 引擎进行播放，代码如下：

```
#include "log.h"
#include "FFmpegCore.h"
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include "libswresample/swresample.h"
#include "libavutil/samplefmt.h"
#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>

uint8_t *outputBuffer;
size_t outputBufferSize;

AVPacket packet;
int audioStream;
AVFrame *aFrame;
SwrContext *swr;
AVFormatContext *aFormatCtx;
AVCodecContext *aCodecCtx;

int initFFmpeg(int *rate, int *channel, char *url) {

    av_register_all();
    aFormatCtx = avformat_alloc_context();
    LOGD("ffmpeg get url=:%s", url);
    //网络音频流
    char *file_name = url;

    //打开音频文件
    if (avformat_open_input(&aFormatCtx, file_name, NULL, NULL) != 0) {
        LOGE("Couldn't open file:%s\n", file_name);
    }
```




```
        return -1; //返回错误, 打开文件失败
    }

    //检索流信息
    if (avformat_find_stream_info(aFormatCtx, NULL) < 0) {
        LOGE("Couldn't find stream information.");
        return -1;
    }

    //找到第一条音频流
    int i;
    audioStream = -1;
    for (i = 0; i < aFormatCtx->nb_streams; i++) {
        if (aFormatCtx->streams[i]->codec->codec_type == AVMEDIA_TYPE_
AUDIO && audioStream < 0) {
            audioStream = i;
        }
    }
    if (audioStream == -1) {
        LOGE("Couldn't find audio stream!");
        return -1;
    }

    //得到音频流的编解码器上下文环境的指针
    aCodecCtx = aFormatCtx->streams[audioStream]->codec;

    //找到音频流的解码器
    AVCodec *aCodec = avcodec_find_decoder(aCodecCtx->codec_id);
    if (!aCodec) {
        fprintf(stderr, "Unsupported codec!\n");
        return -1;
    }

    if (avcodec_open2(aCodecCtx, aCodec, NULL) < 0) {
        LOGE("Could not open codec.");
        return -1; //返回-1, 表示不能打开编解码器
    }

    aFrame = av_frame_alloc();

    //设置格式转换
    swr = swr_alloc();
    av_opt_set_int(swr, "in_channel_layout", aCodecCtx->channel_layout, 0);
```



```
av_opt_set_int(swr, "out_channel_layout", aCodecCtx->channel_layout, 0);
av_opt_set_int(swr, "in_sample_rate", aCodecCtx->sample_rate, 0);
av_opt_set_int(swr, "out_sample_rate", aCodecCtx->sample_rate, 0);
av_opt_set_sample_fmt(swr, "in_sample_fmt", aCodecCtx->sample_fmt, 0);
av_opt_set_sample_fmt(swr, "out_sample_fmt", AV_SAMPLE_FMT_S16, 0);
swr_init(swr);

//分配 PCM 数据缓存
outputBufferSize = 8196;
outputBuffer = (uint8_t *) malloc(sizeof(uint8_t) * outputBufferSize);

//返回采样率和信道
*rate = aCodecCtx->sample_rate;
*channel = aCodecCtx->channels;
return 0;
}

//获取 PCM 数据, 自动回调获取
int getPCM(void **pcm, size_t *pcmSize) {
    LOGD(">> getPcm");
    while (av_read_frame(aFormatCtx, &packet) >= 0) {

        int frameFinished = 0;
        //判断 Packet (音视频压缩数据) 是否来自音频流
        if (packet.stream_index == audioStream) {
            avcodec_decode_audio4(aCodecCtx, aFrame, &frameFinished, &packet);

            if (frameFinished) {
                //data_size 为音频数据所占的字节数
                int data_size = av_samples_get_buffer_size(
                    aFrame->linesize, aCodecCtx->channels,
                    aFrame->nb_samples, aCodecCtx->sample_fmt, 1);
                LOGD(">> getPcm data_size=%d", data_size);
                //这里进行内存再分配可能存在问题
                if (data_size > outputBufferSize) {
                    outputBufferSize = data_size;
                    outputBuffer = (uint8_t *) realloc(outputBuffer,
                        sizeof(uint8_t) * outputBufferSize);
                }

                //音频格式转换
                swr_convert(swr, &outputBuffer, aFrame->nb_samples,
                    (uint8_t const **) (aFrame->extended_data),
                    aFrame->nb_samples);
            }
        }
    }
}
```




```
        //返回 PCM 数据
        *pcm = outputBuffer;
        *pcmSize = data_size;
        return 0;
    }
}
return -1;
}

//释放相关资源
int releaseFFmpeg()
{
    av_packet_unref(&packet);
    av_free(outputBuffer);
    av_free(aFrame);
    avcodec_close(aCodecCtx);
    avformat_close_input(&aFormatCtx);
    return 0;
}
```

NativePlayer.c 类对应 Java 层的 NativePlayer.java 类，获取从 Java 层传入的参数，代码如下：

```
#include "log.h"
#include "com_hejunlin_ffmpegaudio_NativePlayer.h"
#include "OpenSL_ES_Core.h"

JNIEXPORT void JNICALL
Java_com_hejunlin_ffmpegaudio_NativePlayer_play(JNIEnv *env, jclass type,
jstring url_) {
    const char *url = (*env)->GetStringUTFChars(env, url_, 0);
    LOGD("start playaudio... url=%s", url);

    play(url);
    (*env)->ReleaseStringUTFChars(env, url_, url);
}

JNIEXPORT void JNICALL
Java_com_hejunlin_ffmpegaudio_NativePlayer_stop(JNIEnv *env, jclass type)
{
    LOGD("stop");
    stop();
}
```


}

通过 CMake 或 ndk-build 都可以编译库文件，会生成一个 NativePlayer.so 文件，如图 8-13 所示。

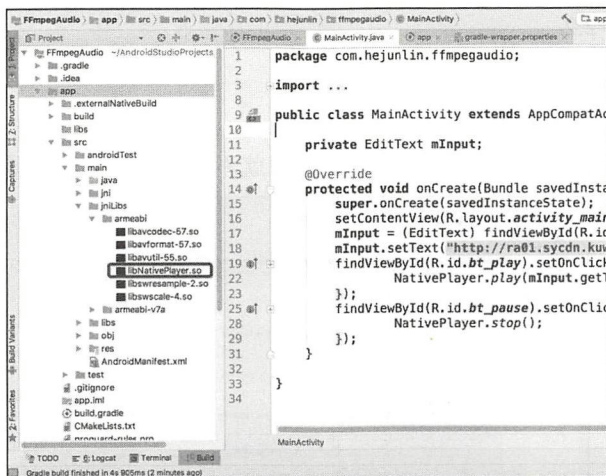


图 8-13 音频解码工程结构图

效果图如图 8-14 所示。

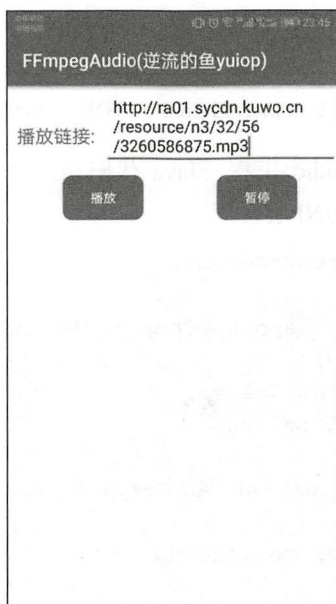


图 8-14 音频解码运行效果图

Android 音视频开发

8.3.5 FFmpeg 视频解码

运行环境：

- Mac OS
- Android Studio 2.2
- android-ndk-r10e
- FFmpeg 3.1.3

Android Studio + NDK 的环境配置很简单，这里就不详细介绍了，如图 8-15 所示。

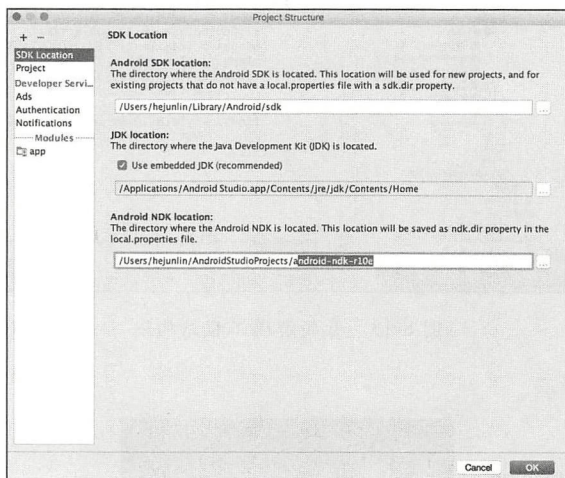


图 8-15 Android Studio+NDK 的环境配置

这个工程同样是 Android Studio 工程，Java 代码如下（主要是设置一些用于展示视频的 SurfaceView，以及传播地址给 JNI 层）：

```
package com.hejunlin.ffmpegdecoder;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class MainActivity extends AppCompatActivity implements SurfaceHolder.
    Callback{
    private SurfaceHolder mSurfaceHolder;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

```

        setContentView(R.layout.activity_main);
        SurfaceView surfaceView = (SurfaceView) findViewById(R.id.
surface_view);
        mSurfaceHolder = surfaceView.getHolder();
        mSurfaceHolder.addCallback(this);
    }

    @Override
    public void surfaceCreated(SurfaceHolder surfaceHolder) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                String url = "rtsp://184.72.239.149/vod/mp4://BigBuck-
Bunny_175k.mov";
                NativePlayer.playVideo(url, mSurfaceHolder.getSurface());
            }
        }).start();
    }

    @Override
    public void surfaceChanged(SurfaceHolder surfaceHolder, int i, int
i1, int i2) {
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder surfaceHolder) {
    }
}

```

NativePlayer.java 主要用于调用 native 函数中的播放视频接口，代码如下：

```

package com.hejunlin.ffmpegdecoder;
public class NativePlayer {
    static {
        System.loadLibrary("yuiopffmpeg");
    }

    public static native int playVideo(String url, Object surface);
}

```

生成头文件 `javah -d jni com.hejunlin.ffmpegdecoder.NativePlayer`，如图 8-16 所示。

复制 FFmpeg-3.1.3 中 android 目录下的 include 文件夹中的文件，其中包括 5 个子文件夹，以及对应的 5 个 .so 动态库文件），生成 include、prebuilt 目录，也可以自己建立文件夹。创建好的目录层级如图 8-17 所示。

Android 音视频开发

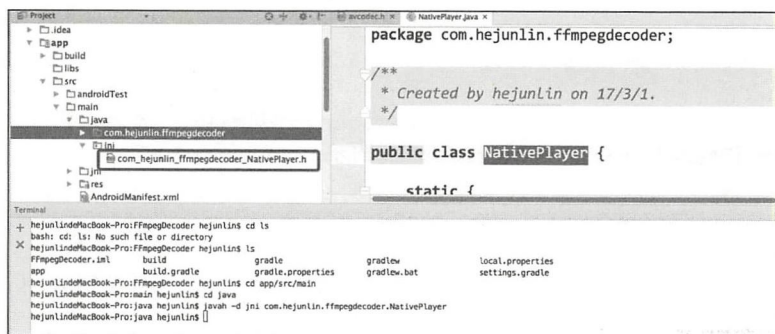


图 8-16 编译生成 NativePlayer 对应的 JNI 头文件

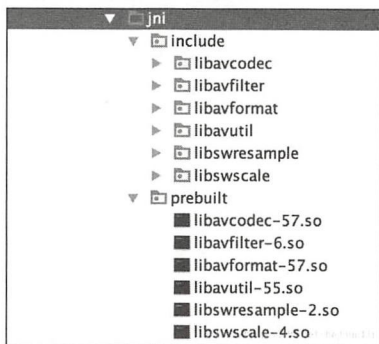


图 8-17 移植 FFmpeg 编译好的头文件和动态库

然后写 make 文件:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := avcodec
LOCAL_SRC_FILES := prebuilt/libavcodec-57.so
include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := avformat
LOCAL_SRC_FILES := prebuilt/libavformat-57.so
include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := avutil
LOCAL_SRC_FILES := prebuilt/libavutil-55.so
include $(PREBUILT_SHARED_LIBRARY)
```

第8章 FFmpeg 源码分析及实战

```
include $(CLEAR_VARS)
LOCAL_MODULE := swresample
LOCAL_SRC_FILES := prebuilt/libswresample-2.so
include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := swscale
LOCAL_SRC_FILES := prebuilt/libswscale-4.so
include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)

LOCAL_SRC_FILES := yuiopffmpeg.c
LOCAL_LDLIBS += -llog -lz -landroid
LOCAL_MODULE := yuiopffmpeg
LOCAL_C_INCLUDES += $(LOCAL_PATH)/include

LOCAL_SHARED_LIBRARIES:= avcodec avformat avutil swresample swscale
include $(BUILD_SHARED_LIBRARY)
```

同时写 `Application.mk`，其主要用于编译不同平台的库。由于引用了 `native_window.h`，故要加上一句 `APP_PLATFORM := android-10`：

```
APP_ABI := armeabi armeabi-v7a x86
APP_PLATFORM := android-10
```

接下来，最主要的工作就是写 JNI 相关代码，`yuiopffmpeg.c` 文件中是实现逻辑，代码如下：

```
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include <android/native_window.h>
#include <android/native_window_jni.h>
#include "com_hejunlin_ffmpegdecoder_NativePlayer.h"
#include "log.h"

JNIEXPORT jint JNICALL Java_com_hejunlin_ffmpegdecoder_NativePlayer_
playVideo (JNIEnv * env, jclass clazz, jstring url, jobject surface)
{
    LOGD("start playvideo... url");

    //视频URL
    const char * file_name = (*env)->GetStringUTFChars(env, url, JNI_FALSE);
    LOGD("start playvideo... url, %s", file_name);
```


Android 音视频开发

```

av_register_all();

AVFormatContext * pFormatCtx = avformat_alloc_context();

//打开视频文件
if(avformat_open_input(&pFormatCtx, file_name, NULL, NULL)!=0) {

    LOGE("Couldn't open file:%s\n", file_name);
    return -1; //打开文件失败, 返回-1
}

//检索流信息
if(avformat_find_stream_info(pFormatCtx, NULL)<0) {
    LOGE("Couldn't find stream information.");
    return -1;
}

//找到第一个视频流
int videoStream = -1, i;
for (i = 0; i < pFormatCtx->nb_streams; i++) {
    if (pFormatCtx->streams[i]->codec->codec_type == AVMEDIA_TYPE_
VIDEO && videoStream < 0) {
        videoStream = i;
    }
}
if(videoStream==-1) {
    LOGE("Didn't find a video stream.");
    return -1; //找不到视频流, 返回-1
}

//得到视频流的编解码器上下文环境的指针
AVCodecContext * pCodecCtx = pFormatCtx->streams[videoStream]->codec;

//找到视频流对应的解码器
AVCodec * pCodec = avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL) {
    LOGE("Codec not found.");
    return -1; //找不到视频解码器, 返回-1
}

if(avcodec_open2(pCodecCtx, pCodec, NULL) < 0) {
    LOGE("Could not open codec.");
}

```



```

        return -1; //打开解码器失败, 返回-1
    }

    //获取 NativeWindow, 用于渲染视频
    ANativeWindow* nativeWindow = ANativeWindow_fromSurface(env, surface);

    //获取视频宽高值
    int videoWidth = pCodecCtx->width;
    int videoHeight = pCodecCtx->height;

    //设置 NativeWindow 的 Buffer 大小, 可自动拉伸
    ANativeWindow_setBuffersGeometry(nativeWindow, videoWidth, videoHeight,
    WINDOW_FORMAT_RGBA_8888);
    ANativeWindow_Buffer windowBuffer;
    if(avcodec_open2(pCodecCtx, pCodec, NULL)<0) {
        LOGE("Could not open codec.");
        return -1; //打开解码器失败, 返回-1
    }

    //分配视频帧空间内存
    AVFrame * pFrame = av_frame_alloc();
    //用于渲染
    AVFrame * pFrameRGBA = av_frame_alloc();
    if(pFrameRGBA == NULL || pFrame == NULL) {
        LOGE("Could not allocate video frame.");
        return -1;
    }

    //确定所需缓冲区大小并分配缓冲区内存空间
    //Buffer 中的数据就是用于渲染的, 且格式为 RGBA
    int numBytes=av_image_get_buffer_size(AV_PIX_FMT_RGBA, pCodecCtx->
width, pCodecCtx->height, 1);
    uint8_t * buffer=(uint8_t *)av_malloc(numBytes*sizeof(uint8_t));
    av_image_fill_arrays(pFrameRGBA->data, pFrameRGBA->linesize, buffer,
    AV_PIX_FMT_RGBA, pCodecCtx->width, pCodecCtx->height, 1);

    //由于解码出来的帧格式不是 RGBA 的, 故在渲染之前需要进行格式转换
    struct SwsContext *sws_ctx = sws_getContext(pCodecCtx->width,
        pCodecCtx->height,
        pCodecCtx->pix_fmt,
        pCodecCtx->width,
        pCodecCtx->height,
        AV_PIX_FMT_RGBA,
        SWS_BILINEAR,

```

Android 音视频开发

```

        NULL,
        NULL,
        NULL);

int frameFinished;
AVPacket packet;
while(av_read_frame(pFormatCtx, &packet)>=0) {
    //判断 Packet (音视频压缩数据) 是否是视频流
    if(packet.stream_index==videoStream) {
        //解码视频帧
        avcodec_decode_video2(pCodecCtx, pFrame, &frameFinished, &packet);

        //并不是解码一次就可以解码出 1 帧
        if (frameFinished) {

            //锁住 NativeWindow 的缓冲区
            ANativeWindow_lock(nativeWindow, &windowBuffer, 0);

            //格式转换
            sws_scale(sws_ctx, (uint8_t const * const *)pFrame->data,
                pFrame->linesize, 0, pCodecCtx->height,
                pFrameRGBA->data, pFrameRGBA->linesize);

            //获取 stride
            uint8_t * dst = windowBuffer.bits;
            int dstStride = windowBuffer.stride * 4;
            uint8_t * src = (uint8_t*) (pFrameRGBA->data[0]);
            int srcStride = pFrameRGBA->linesize[0];
            //由于窗口的 stride 和帧的 stride 不同, 因此需要逐行复制
            int h;
            for (h = 0; h < videoHeight; h++) {
                memcpy(dst + h * dstStride, src + h * srcStride,
srcStride);
            }

            ANativeWindow_unlockAndPost(nativeWindow);
        }
    }
    av_packet_unref(&packet);
}

av_free(buffer);
av_free(pFrameRGBA);

```



```

//释放 YUV 图像帧
av_free(pFrame);
//关闭解码器
avcodec_close(pCodecCtx);
//关闭视频文件
avformat_close_input(&pFormatCtx);
return 0;
}

```

这时就可以执行编译操作了，完成后能发现多了一个 so 动态库文件，即 libyuiopffmpegs.o 文件，这是编译出来的文件，如图 8-18 所示。

这时候，在 Android Studio 的 main 目录下，建立一个 jniLibs，把刚刚那个 so 文件复制到这里就行了。不同平台，如 ARM、X86，相应文件夹的名字不同，也不能随便更改。记得还要在 build.gradle 中进行如图 8-19 所示的配置。

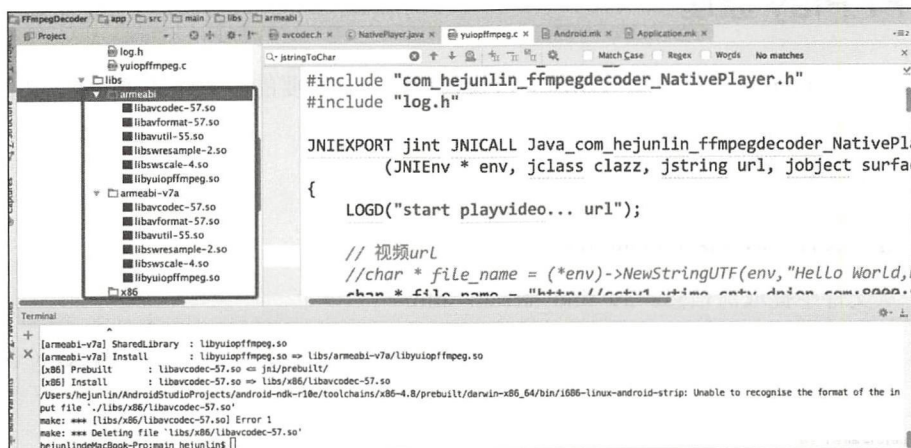


图 8-18 编译出新的动态库文件

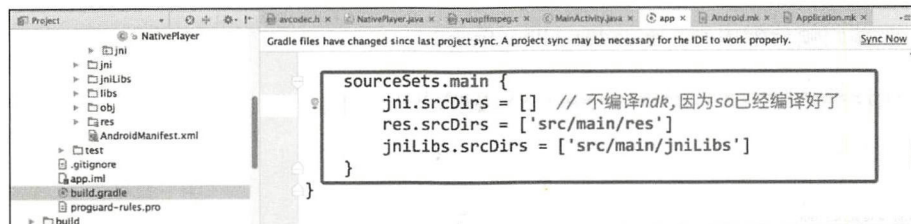


图 8-19 在 build.gradle 文件中进行配置

最后的效果如图 8-20 所示。



图 8-20 FFmpeg 进行视频解码的最终效果

8.4 FFPlay 原理

FFPlay 是 FFmpeg 中拥有播放功能的可执行程序，它主要的源文件是 `ffplay.c` 文件，在该文件中播放一个视频的步骤如下。

- (1) 注册所有容器格式和 Codec: `av_register_all`。
- (2) 打开文件: `av_open_input_file`。
- (3) 从文件中提取流信息: `av_find_stream_info`。
- (4) 遍历所有的数据流，查找其中的数据流种类: `CODEC_TYPE_VIDEO`（表示视频流）。
- (5) 查找对应的解码器: `avcodec_find_decoder`。
- (6) 打开编解码器: `avcodec_open`。
- (7) 为解码帧分配内存: `avcodec_alloc_frame`。
- (8) 不停地从码流中提取出帧数据: `av_read_frame`。
- (9) 判断帧的类型，调用视频帧: `avcodec_decode_video`。
- (10) 解码完成后，释放解码器: `avcodec_close`。
- (11) 关闭输入文件: `av_close_input_file`。

8.4.1 注册所有容器格式和 Codec

ffplay.c 中的 main 函数，第一步就是调用 av_register_all 函数，我们看看此函数的实现代码：

```
void av_register_all(void)
{
    static int initialized;
    if (initialized)
        return;
    avcodec_register_all();
    /* (de)muxers */
    REGISTER_MUXER    (A64,          a64);
    REGISTER_DEMUXER  (AA,           aa);
    REGISTER_DEMUXER  (AAC,          aac);
    REGISTER_MUXDEMUX(AC3,          ac3);
    REGISTER_DEMUXER  (ACM,          acm);
    REGISTER_DEMUXER  (ACT,          act);
    REGISTER_DEMUXER  (ADF,          adf);
    REGISTER_DEMUXER  (ADP,          adp);
    REGISTER_DEMUXER  (ADS,          ads);
    REGISTER_MUXER    (ADTS,         adts);
    REGISTER_MUXDEMUX(ADX,          adx);
    //省略注册各种复用器及解复用器的代码部分
    initialized = 1;
}
```

av_register_all 内部会调用 avcodec_register_all、av_register_output_format 及 av_register_input_format 函数，即注册所有编解码器、解复用器、协议。

8.4.2 打开流文件

在初始化一些结构体后，开始调用 is = stream_open(input_filename, file_iformat);，下面看看这个函数做了什么，代码如下：

```
static VideoState *stream_open(const char *filename, AVInputFormat *iformat)
{
    VideoState *is;
    is = av_mallocz(sizeof(VideoState));
    if (!is)
        return NULL;
    is->filename = av_strdup(filename);
    if (!is->filename)
        goto fail;
}
```

```

        is->iformat = iformat;
        is->ytop     = 0;
        is->xleft    = 0;

        /*显示视频图像*/
        if (frame_queue_init(&is->pictq, &is->videoq, VIDEO_PICTURE_QUEUE_
SIZE, 1) < 0)
            goto fail;
        if (frame_queue_init(&is->subpq, &is->subtitleq, SUBPICTURE_QUEUE_
SIZE, 0) < 0)
            goto fail;
        if (frame_queue_init(&is->sampq, &is->audioq, SAMPLE_QUEUE_SIZE, 1) < 0)
            goto fail;

        if (packet_queue_init(&is->videoq) < 0 ||
            packet_queue_init(&is->audioq) < 0 ||
            packet_queue_init(&is->subtitleq) < 0)
            goto fail;

        if (!(is->continue_read_thread = SDL_CreateCond())) {
            av_log(NULL, AV_LOG_FATAL, "SDL_CreateCond(): %s\n", SDL_GetError());
            goto fail;
        }

        init_clock(&is->vidclk, &is->videoq.serial);
        init_clock(&is->audclk, &is->audioq.serial);
        init_clock(&is->extclk, &is->extclk.serial);
        is->audio_clock_serial = -1;
        is->audio_volume = SDL_MIX_MAXVOLUME;
        is->muted = 0;
        is->av_sync_type = av_sync_type;
        is->read_tid     = SDL_CreateThread(read_thread, is);
        if (!is->read_tid) {
            av_log(NULL, AV_LOG_FATAL, "SDL_CreateThread(): %s\n", SDL_GetError());
fail:
            stream_close(is);
            return NULL;
        }
        return is;
    }

```

在这里会初始化音频、视频时钟，最主要的是创建一个读流内容的线程，也就是 `SDL_CreateThread(read_thread, is);`。

8.4.3 读取数据

读取数据主要是通过 `read_thread` 函数进行的。在 `read_thread` 函数中，可以看到如下代码：

```
static int read_thread(void *arg)
{
    //省略部分代码
    memset(st_index, -1, sizeof(st_index));
    is->last_video_stream = is->video_stream = -1;
    is->last_audio_stream = is->audio_stream = -1;
    is->last_subtitle_stream = is->subtitle_stream = -1;
    is->eof = 0;

    ic = avformat_alloc_context(); //1.分配 AVFormatContext
    if (!ic) {
        av_log(NULL, AV_LOG_FATAL, "Could not allocate context.\n");
        ret = AVERROR(ENOMEM);
        goto fail;
    }
    ic->interrupt_callback.callback = decode_interrupt_cb;
    //2.设置解码中断 callback
    ic->interrupt_callback.opaque = is;
    if (!av_dict_get(format_opts, "scan_all_pmts", NULL, AV_DICT_MATCH_
CASE)) {
        av_dict_set(&format_opts, "scan_all_pmts", "1", AV_DICT_DONT_
OVERWRITE);
        scan_all_pmts_set = 1;
    }
    err = avformat_open_input(&ic, is->filename, is->iformat, &format_opts);
    //3.打开文件
    if (err < 0) {
        print_error(is->filename, err);
        ret = -1;
        goto fail;
    }
    //省略部分代码
    is->ic = ic;

    if (genpts)
        ic->flags |= AVFMT_FLAG_GENPTS;

    av_format_inject_global_side_data(ic);

    opts = setup_find_stream_info_opts(ic, codec_opts);
```

```

orig_nb_streams = ic->nb_streams;

err = avformat_find_stream_info(ic, opts); //4. 查找音频流、视频流、字幕流

for (i = 0; i < orig_nb_streams; i++)
    av_dict_free(&opts[i]);
av_freep(&opts);

//省略部分代码
is->realtime = is_realtime(ic);
if (show_status)
    av_dump_format(ic, 0, is->filename, 0);
//省略部分代码
is->show_mode = show_mode;
//省略部分代码

/*打开数据流*/
if (st_index[AVMEDIA_TYPE_AUDIO] >= 0) { //5. 如果有音频流，则打开音频流
    stream_component_open(is, st_index[AVMEDIA_TYPE_AUDIO]);
}

ret = -1;
if (st_index[AVMEDIA_TYPE_VIDEO] >= 0) { //6. 如果有视频流，则打开视频流
    ret = stream_component_open(is, st_index[AVMEDIA_TYPE_VIDEO]);
}
if (is->show_mode == SHOW_MODE_NONE)
    is->show_mode = ret >= 0 ? SHOW_MODE_VIDEO : SHOW_MODE_RDFT;
if (st_index[AVMEDIA_TYPE_SUBTITLE] >= 0) { //7. 如果有字幕流，则打开字幕流
    stream_component_open(is, st_index[AVMEDIA_TYPE_SUBTITLE]);
}

if (is->video_stream < 0 && is->audio_stream < 0) {
    av_log(NULL, AV_LOG_FATAL, "Failed to open file '%s' or configure
filtergraph\n", is->filename);
    ret = -1;
    goto fail;
}

if (infinite_buffer < 0 && is->realtime)
    infinite_buffer = 1;

for (;;) {
    if (is->abort_request)
        break;

```



```

        if (is->paused != is->last_paused) {
            is->last_paused = is->paused;
            if (is->paused)
                is->read_pause_return = av_read_pause(ic);
            else
                av_read_play(ic);
        }
#ifdef CONFIG_RTSP_DEMUXER || CONFIG_MMSH_PROTOCOL
        if (is->paused &&
            (!strcmp(ic->iformat->name, "rtsp") ||
             (ic->pb && !strncmp(input_filename, "mmsh:", 5)))) {
            //等待10ms, 避免立马尝试获取下一个 Packet
            SDL_Delay(10);
            continue;
        }
#endif
        if (is->seek_req) {
            int64_t seek_target = is->seek_pos;
            int64_t seek_min = is->seek_rel > 0 ? seek_target - is->seek_rel
+ 2: INT64_MIN;
            int64_t seek_max = is->seek_rel < 0 ? seek_target - is->seek_rel
- 2: INT64_MAX;
            //修复由于四舍五入, 没有在 seek_pos/seek_rel 变量的正确方向上进行
            ret = avformat_seek_file(is->ic, -1, seek_min, seek_target,
seek_max, is->seek_flags);
            if (ret < 0) {
                av_log(NULL, AV_LOG_ERROR,
                    "%s: error while seeking\n", is->ic->filename);
            } else {
                if (is->audio_stream >= 0) {
                    packet_queue_flush(&is->audioq); //音频流入队音频队列
                    packet_queue_put(&is->audioq, &flush_pkt);
                }
                if (is->subtitle_stream >= 0) {
                    packet_queue_flush(&is->subtitleq);
                    packet_queue_put(&is->subtitleq, &flush_pkt);
                }
                if (is->video_stream >= 0) { //视频流入队视频队列
                    packet_queue_flush(&is->videoq);
                    packet_queue_put(&is->videoq, &flush_pkt);
                }
                if (is->seek_flags & AVSEEK_FLAG_BYTE) {
                    set_clock(&is->extclk, NAN, 0);
                } else {

```



```

        set_clock(&is->extclk, seek_target / (double)AV_TIME_
BASE, 0);

    }

}

is->seek_req = 0;
is->queue_attachments_req = 1;
is->eof = 0;
if (is->paused)
    step_to_next_frame(is);
}
//省略部分代码
ret = av_read_frame(ic, pkt); //不停地从码流中提取出帧数据
//省略部分代码
//检查数据包是否在用户指定的播放范围内，是的话就入队，否则就丢弃
stream_start_time = ic->streams[pkt->stream_index]->start_time;
pkt_ts = pkt->pts == AV_NOPTS_VALUE ? pkt->dts : pkt->pts;
pkt_in_play_range = duration == AV_NOPTS_VALUE || (pkt_ts -
(stream_start_time != AV_NOPTS_VALUE ? stream_start_time : 0)) * av_q2d(ic-
>streams[pkt->stream_index]->time_base) - (double)(start_time != AV_NOPTS_
VALUE ? start_time : 0) / 1000000 <= ((double)duration / 1000000);
if (pkt->stream_index == is->audio_stream && pkt_in_play_range)
{
    packet_queue_put(&is->audioq, pkt);
} else if (pkt->stream_index == is->video_stream && pkt_in_play_
_range && !(is->video_st->disposition & AV_DISPOSITION_ATTACHED_PIC)) {
    packet_queue_put(&is->videoq, pkt);
} else if (pkt->stream_index == is->subtitle_stream && pkt_in_
play_range) {
    packet_queue_put(&is->subtitleq, pkt);
} else {
    av_packet_unref(pkt);
}
}

ret = 0;
//省略部分代码
return 0;
}

```

其中，`stream_component_open` 函数用于打开视频流或音频流，开始寻找解码器解码数据。`stream_component_open` 函数的代码如下：

```

static int stream_component_open(VideoState *is, int stream_index)
{
    AVFormatContext *ic = is->ic;

```

```

AVCodecContext *avctx;
AVCodec *codec;
const char *forced_codec_name = NULL;
AVDictionary *opts = NULL;
AVDictionaryEntry *t = NULL;
int sample_rate, nb_channels;
int64_t channel_layout;
int ret = 0;
int stream_lowres = lowres;
if (stream_index < 0 || stream_index >= ic->nb_streams)
    return -1;

avctx = avcodec_alloc_context3(NULL);
if (!avctx)
    return AVERROR(ENOMEM);
ret = avcodec_parameters_to_context(avctx, ic-> streams[stream_index]
->codecpar);
if (ret < 0)
    goto fail;
av_codec_set_pkt_timebase(avctx, ic->streams[stream_index]->time_base);
//1.设置 Packet(音视频压缩数据)的 timebase
codec = avcodec_find_decoder(avctx->codec_id);
//2.通过 codec_id 找解码器
switch(avctx->codec_type){//将解码器类型、视频、音频、字幕分开
    case AVMEDIA_TYPE_AUDIO : is->last_audio_stream = stream_index;
        forced_codec_name = audio_codec_name; break;
    case AVMEDIA_TYPE_SUBTITLE: is->last_subtitle_stream = stream_index;
        forced_codec_name = subtitle_codec_name; break;
    case AVMEDIA_TYPE_VIDEO : is->last_video_stream = stream_index;
        forced_codec_name = video_codec_name; break;
}
if (forced_codec_name)
    codec = avcodec_find_decoder_by_name(forced_codec_name);
if (!codec) {
    if (forced_codec_name) av_log(NULL, AV_LOG_WARNING, "No codec
could be found with name '%s'\n", forced_codec_name);
    else av_log(NULL, AV_LOG_WARNING, "No codec could be found with
id %d\n", avctx->codec_id);
    ret = AVERROR(EINVAL);
    goto fail;
}

avctx->codec_id = codec->id;
if(stream_lowres > av_codec_get_max_lowres(codec)){

```



```

        av_log(avctx, AV_LOG_WARNING, "The maximum value for lowres
supported by the decoder is %d\n", av_codec_get_max_lowres(codec));
        stream_lowres = av_codec_get_max_lowres(codec);
    }
    av_codec_set_lowres(avctx, stream_lowres);

#ifdef FF_API_EMU_EDGE
    if(stream_lowres) avctx->flags |= CODEC_FLAG_EMU_EDGE;
#endif
    if (fast)
        avctx->flags2 |= AV_CODEC_FLAG2_FAST;
#ifdef FF_API_EMU_EDGE
    if(codec->capabilities & AV_CODEC_CAP_DR1)
        avctx->flags |= CODEC_FLAG_EMU_EDGE;
#endif

    opts = filter_codec_opts(codec_opts, avctx->codec_id, ic, ic->streams
[stream_index], codec);
    if (!av_dict_get(opts, "threads", NULL, 0))
        av_dict_set(&opts, "threads", "auto", 0);
    if (stream_lowres)
        av_dict_set_int(&opts, "lowres", stream_lowres, 0);
    if (avctx->codec_type == AVMEDIA_TYPE_VIDEO || avctx->codec_type ==
AVMEDIA_TYPE_AUDIO)
        av_dict_set(&opts, "refcounted_frames", "1", 0);
    if ((ret = avcodec_open2(avctx, codec, &opts)) < 0) {
        goto fail;
    }
    if ((t = av_dict_get(opts, "", NULL, AV_DICT_IGNORE_SUFFIX))) {
        av_log(NULL, AV_LOG_ERROR, "Option %s not found.\n", t->key);
        ret = AERROR_OPTION_NOT_FOUND;
        goto fail;
    }

    is->eof = 0;
    ic->streams[stream_index]->discard = AVDISCARD_DEFAULT;
    switch (avctx->codec_type) {
        case AVMEDIA_TYPE_AUDIO:
#ifdef CONFIG_AVFILTER
            {
                AVFilterLink *link;

                is->audio_filter_src.freq          = avctx->sample_rate;
                is->audio_filter_src.channels       = avctx->channels;

```



```

        is->audio_filter_src.channel_layout = get_valid_channel_
layout(avctx->channel_layout, avctx->channels);
        is->audio_filter_src.fmt           = avctx->sample_fmt;
        if ((ret = configure_audio_filters(is, afilters, 0)) < 0)
            goto fail;
        link = is->out_audio_filter->inputs[0];
        sample_rate      = link->sample_rate;
        nb_channels      = avfilter_link_get_channels(link);
        channel_layout   = link->channel_layout;
    }
#else
    sample_rate      = avctx->sample_rate;
    nb_channels      = avctx->channels;
    channel_layout   = avctx->channel_layout;
#endif

    /*准备音频输出*/
    if ((ret = audio_open(is, channel_layout, nb_channels, sample_
rate, &is->audio_tgt)) < 0) //准备音频输出
        goto fail;
    is->audio_hw_buf_size = ret;
    is->audio_src = is->audio_tgt;
    is->audio_buf_size = 0;
    is->audio_buf_index = 0;

    /*初始化 averaging 滤镜 */
    is->audio_diff_avg_coef = exp(log(0.01) / AUDIO_DIFF_AVG_NB);
    is->audio_diff_avg_count = 0;
    /*由于我们没有精确的音频数据充满 FIFO(先进先出队列)，故只有在大于这个阈值时
才校正音频同步*/
    is->audio_diff_threshold = (double)(is->audio_hw_buf_size) / is->
audio_tgt.bytes_per_sec;
    is->audio_stream = stream_index;
    is->audio_st = ic->streams[stream_index];
    decoder_init(&is->auddec, avctx, &is->audioq, is->continue_read_
thread);
    if ((is->ic->iformat->flags & (AVFMT_NOBINSEARCH | AVFMT_NOGENSEARCH
| AVFMT_NO_BYTE_SEEK)) && !is->ic->iformat->read_seek) {
        is->auddec.start_pts = is->audio_st->start_time;
        is->auddec.start_pts_tb = is->audio_st->time_base;
    }
    if ((ret = decoder_start(&is->auddec, audio_thread, is)) < 0)
        goto out;
    SDL_PauseAudio(0);

```

```

        break;
    case AVMEDIA_TYPE_VIDEO:
        is->video_stream = stream_index;
        is->video_st = ic->streams[stream_index];
        is->viddec_width = avctx->width;
        is->viddec_height = avctx->height;

        decoder_init(&is->viddec, avctx, &is->videoq, is->continue_read_
thread);
        if ((ret = decoder_start(&is->viddec, video_thread, is)) < 0)
            goto out;
        is->queue_attachments_req = 1;
        break;
    case AVMEDIA_TYPE_SUBTITLE:
        is->subtitle_stream = stream_index;
        is->subtitle_st = ic->streams[stream_index];

        decoder_init(&is->subdec, avctx, &is->subtitleq, is->continue_read_
thread);
        if ((ret = decoder_start(&is->subdec, subtitle_thread, is)) < 0)
            goto out;
        break;
    default:
        break;
}
goto out;

fail:
    avcodec_free_context(&avctx);
out:
    av_dict_free(&opts);

    return ret;
}

```

从以上代码中可以看出，音频、视频、字幕分别有自己的线程进行解码，也就是 video_thread、audio_thread、subtitle_thread。

8.4.4 保存数据

当我们准备好解码器后，可以开始执行视频解码线程 video_thread。这个线程从视频队列中读取包，把它解码成视频帧，然后调用 queue_picture 函数把处理好的帧放入图像队列中，代码如下：


```

static int video_thread(void *arg)
{
    VideoState *is = arg;
    AVFrame *frame = av_frame_alloc(); //分配解码后的帧数据
    double pts;
    double duration;
    int ret;
    AVRational tb = is->video_st->time_base;
    AVRational frame_rate = av_guess_frame_rate(is->ic, is->video_st, NULL);

    if (!frame) {
#ifdef CONFIG_AVFILTER
        avfilter_graph_free(&graph);
#endif
        return AVERROR(ENOMEM);

        for (;;) { //循环取出解码的帧数据
            ret = get_video_frame(is, frame);
            if (ret < 0)
                goto the_end;
            if (!ret)
                continue;
#ifdef CONFIG_AVFILTER
            if (last_w != frame->width
                || last_h != frame->height
                || last_format != frame->format
                || last_serial != is->viddec.pkt_serial
                || last_vfilter_idx != is->vfilter_idx) {
                av_log(NULL, AV_LOG_DEBUG, "Video frame changed from size: %dx%d
format:%s serial:%d to size:%dx%d format:%s serial:%d\n", last_w, last_h, (const
char *)av_x_if_null(av_get_pix_fmt_name(last_format), "none"), last_serial,
frame-> width, frame->height, (const char *)av_x_if_null(av_get_pix_fmt_
name(frame->format), "none"), is->viddec.pkt_serial);
                avfilter_graph_free(&graph);
                graph = avfilter_graph_alloc();
                if ((ret = configure_video_filters(graph, is, vfilters_list ?
vfilters_list[is->vfilter_idx] : NULL, frame)) < 0) {
                    SDL_Event event;
                    event.type = FF_QUIT_EVENT;
                    event.user.data1 = is;
                    SDL_PushEvent(&event);
                    goto the_end;
                }
            }
            filt_in = is->in_video_filter;

```



```

        filt_out = is->out_video_filter;
        last_w = frame->width;
        last_h = frame->height;
        last_format = frame->format;
        last_serial = is->viddec.pkt_serial;
        last_vfilter_idx = is->vfilter_idx;
        frame_rate = filt_out->inputs[0]->frame_rate;
    }
    ret = av_buffersrc_add_frame(filt_in, frame);
    if (ret < 0)
        goto the_end;
    while (ret >= 0) {
        is->frame_last_returned_time = av_gettime_relative() / 1000000.0;
        ret = av_buffersink_get_frame_flags(filt_out, frame, 0);
        if (ret < 0) {
            if (ret == AERROR_EOF)
                is->viddec.finished = is->viddec.pkt_serial;
            ret = 0;
            break;
        }
        is->frame_last_filter_delay = av_gettime_relative() / 1000000.0 -
is->frame_last_returned_time;
        if (fabs(is->frame_last_filter_delay) > AV_NOSYNC_THRESHOLD
/ 10.0) is->frame_last_filter_delay = 0;
        tb = filt_out->inputs[0]->time_base;

        #endif

        duration = (frame_rate.num && frame_rate.den ? av_
q2d((AVRational){frame_rate.den, frame_rate.num}) : 0);
        pts = (frame->pts == AV_NOPTS_VALUE) ? NAN : frame->pts * av_
q2d(tb);

        ret = queue_picture(is, frame, pts, duration, av_frame_get_
pkt_pos(frame), is->viddec.pkt_serial);
        //把 Frame 的 PTS、duration、pkt_serial 塞入图像队列中
        av_frame_unref(frame);
        #if CONFIG_AVFILTER
    }
        #endif

        if (ret < 0)
            goto the_end;
    }
    the_end:
    #if CONFIG_AVFILTER
        avfilter_graph_free(&graph);
    #endif
}

```

```

#endif
    av_frame_free(&frame);
    return 0;
}

```

audio_thread 函数是音频解码线程的入口函数，代码如下：

```

static int audio_thread(void *arg)
{
    VideoState *is = arg;
    AVFrame *frame = av_frame_alloc();
    Frame *af;
#ifdef CONFIG_AVFILTER
    int last_serial = -1;
    int64_t dec_channel_layout;
    int reconfigure;
#endif
    int got_frame = 0;
    AVRational tb;
    int ret = 0;

    if (!frame)
        return AVERERROR(ENOMEM);

    do {
        if ((got_frame = decoder_decode_frame(&is->auddec, frame, NULL)) < 0)
            goto the_end;
        if (got_frame) {
            tb = (AVRational){1, frame->sample_rate};

#ifdef CONFIG_AVFILTER
            dec_channel_layout = get_valid_channel_layout(frame->
channel_layout, av_frame_get_channels(frame));
            //省略部分代码
            if ((ret = av_buffersrc_add_frame(is->in_audio_filter, frame)) <
0) goto the_end;
            while ((ret = av_buffersink_get_frame_flags(is->out_audio_
filter, frame, 0)) >= 0) {
                tb = is->out_audio_filter->inputs[0]->time_base;
#endif

                if (!(af = frame_queue_peek_writable(&is->sampq)))
                    goto the_end;
                af->pts = (frame->pts == AV_NOPTS_VALUE) ? NAN : frame->
pts * av_q2d(tb); //对应 PTS
                af->pos = av_frame_get_pkt_pos(frame); //对应 position

```



```

        af->serial = is->auddec.pkt_serial;
        af->duration = av_q2d(AVRational){frame->nb_samples,
frame->sample_rate}); //对应 duration

        av_frame_move_ref(af->frame, frame);
        frame_queue_push(&is->sampq); //入音频 sample 队列

    #if CONFIG_AVFILTER
        if (is->audioq.serial != is->auddec.pkt_serial)
            break;
    }
    if (ret == AVERERROR_EOF)
        is->auddec.finished = is->auddec.pkt_serial;
    #endif
}
} while (ret >= 0 || ret == AVERERROR(EAGAIN) || ret == AVERERROR_EOF);
the_end:
#if CONFIG_AVFILTER
    avfilter_graph_free(&is->agraph);
#endif
    av_frame_free(&frame);
    return ret;
}

```

从上面的代码可以看出，一开始就进入循环，然后调用 `decoder_decode_frame` 函数进行解码，解码后的帧存放到 `Frame` 中，接着调用 `frame_queue_peek_writable` 函数判断是否能把刚刚解码的 `Frame` 写入 `is->sampq` 中，因为 `is->sampq` 是音频解码帧列表，所以播放线程直接从这里读取数据并播放出来。最后 `av_frame_move_ref(af->frame, frame);` 把 `Frame` 放入 `sampq` 的相应位置。由于前面的 `af = frame_queue_peek_writable(&is->sampq)`，`af` 就是指向这一帧 `Frame` 应该放的位置的指针，所以直接把值赋给其结构体里面的 `Frame` 就行了。

`frame_queue_push(&is->sampq);` 里面包含一个唤醒线程的操作，如果音频播放线程因为 `sampq` 队列为空而阻塞，那么在这里可以唤醒它。`decoder_decode_frame` 里调用了传进去的 `Codec` 的 `codec->decode` 函数解码。

在 `frame_queue_peek_writable` 函数里面会判断 `sampq` 队列是否满了，如果没位置放我们的 `Frame`，会调用 `pthread_cond_wait` 函数阻塞队列；如果有位置放 `Frame`，就会返回 `Frame` 应该放置位置的地址。解码线程到此就结束了。

8.4.5 音视频同步

`main` 函数的最后，会调用 `event_loop(is)` 函数。这个函数会进行消息轮询，一方面处理 UI

上传过来的事件并进行响应，如 play、pause、stop 等；另一方面，不断做图像刷新和音视频同步。前面已经了解到，视频解码后的帧数据会存放到 picture_queue 中，音频解码后的帧数据会存放到 sample_queue 中。下面先来看看 event_loop 函数的代码：

```
static void event_loop(VideoState *cur_stream)
{
    SDL_Event event;
    double incr, pos, frac;

    for (;;) {
        double x;
        refresh_loop_wait_event(cur_stream, &event);
        //省略部分代码
    }
}

static void refresh_loop_wait_event(VideoState *is, SDL_Event *event) {
    double remaining_time = 0.0;
    SDL_PumpEvents();
    while (!SDL_PeepEvents(event, 1, SDL_GETEVENT, SDL_ALLEVENTS)) {
        //省略部分代码
        remaining_time = REFRESH_RATE;
        if (is->show_mode != SHOW_MODE_NONE && (!is->paused || is->force_
refresh)) video_refresh(is, &remaining_time);
        SDL_PumpEvents();
    }
}
```

同步的相关逻辑代码如下：

```
static void video_refresh(void *opaque, double *remaining_time)
{
    VideoState *is = opaque;
    double time;
    Frame *sp, *sp2;
    if (!is->paused && get_master_sync_type(is) == AV_SYNC_EXTERNAL_
CLOCK && is->realtime)
        check_external_clock_speed(is);

    if (!display_disable && is->show_mode != SHOW_MODE_VIDEO && is->
audio_st) {
        time = av_gettime_relative() / 1000000.0;
        if (is->force_refresh || is->last_vis_time + rdftspeed < time) {
            video_display(is);
            is->last_vis_time = time;
        }
    }
}
```

```

    }
    *remaining_time = FFMIN(*remaining_time, is->last_vis_time +
rdftspeed - time);
}

    if (is->video_st) {
retry:
        if (frame_queue_nb_remaining(&is->pictq) == 0) {
            //什么也不做, 队列中没有图像展示
        } else {
            double last_duration, duration, delay;
            Frame *vp, *lastvp;
            /*图像出队*/
            lastvp = frame_queue_peek_last(&is->pictq);
            //从队列中取出上一个 Frame
            vp = frame_queue_peek(&is->pictq); //从队列中取出当前 Frame
            /*计算流的持续时长*/
            last_duration = vp_duration(is, lastvp, vp);
            delay = compute_target_delay(last_duration, is); //计算延时
            //lastvp 是上一帧, vp 是当前帧, last_duration 则是根据当前帧和上一帧的 PTS 计算出来的
            //上一帧的显示时间, 经过 compute_target_delay 函数, 计算出显示当前帧需要等待的时间

            time = av_gettime_relative()/1000000.0;
            if (time < is->frame_timer + delay) {
                *remaining_time = FFMIN(is->frame_timer + delay - time,
*remaining_time);
                goto display;
            }
            //frame_timer 实际上就是上一帧的播放时间, 而 frame_timer + delay 实
            //际上是当前这一帧的播放时间, 如果系统时间还没有到当前这一帧的播放时间, 直接跳转至
            //display, 而此时 is->force_refresh 变量为 0, 不显示当前帧, 进入 video_refresh_thread
            //中的下一次循环, 并睡眠等待
            is->frame_timer += delay;
            if (delay > 0 && time - is->frame_timer > AV_SYNC_THRESHOLD_MAX)
                is->frame_timer = time;
            SDL_LockMutex(is->pictq.mutex);
            if (!isnan(vp->pts))
                update_video_pts(is, vp->pts, vp->pos, vp->serial);
            SDL_UnlockMutex(is->pictq.mutex);
            if (frame_queue_nb_remaining(&is->pictq) > 1) {
                Frame *nextvp = frame_queue_peek_next(&is->pictq);
                duration = vp_duration(is, vp, nextvp);
                if (!is->step && (framedrop > 0 || (framedrop && get_master
_sync_type(is) != AV_SYNC_VIDEO_MASTER)) && time > is->frame_timer + duration){

```



```

        is->frame_drops_late++;
        frame_queue_next(&is->pictq);
        goto retry;
    }
}

//如果当前这一帧的播放时间已经过了，并且其和当前系统时间的差值超过了
//AV_SYNC_THRESHOLD_MAX，则将当前这一帧的播放时间改为系统时间，并在后续判断是否
//需要丢帧，其目的是为后面帧的播放时间重新调整 frame_timer，如果缓冲区中有更多的数据，
//并且当前的时间已经大于当前帧的持续显示时间，则丢弃当前帧，尝试显示下一帧
//省略部分代码
frame_queue_next(&is->pictq);
is->force_refresh = 1;
if (is->step && !is->paused)
    stream_toggle_pause(is);
}

display:
    /*显示图像*/
    if (!display_disable && is->force_refresh && is->show_mode ==
SHOW_MODE_VIDEO && is->pictq.rindex_shown)
        video_display(is);
    }
    is->force_refresh = 0;
    //省略部分代码
}

```

对于播放器来说，音视频同步是关键点，同时也是难点，同步效果的好坏直接决定着播放器的质量。通常音视频同步的解决方案就是，选择一个参考时钟，播放时读取音视频帧上的时间戳，同时依照当前参考时钟上的时间戳来安排播放，如图 8-21 所示。

如果音视频帧的播放时间戳大于当前参考时钟上的时间戳，则不急于播放该帧，直到参考时钟达到该帧的时间戳；如果音视频帧的时间戳小于当前参考时钟上的时间戳，则需要“尽快”播放该帧或丢弃，以便播放进度追上参考时钟。

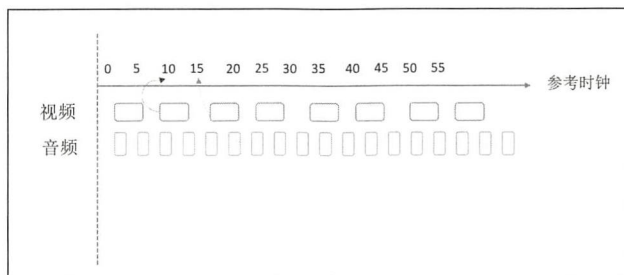


图 8-21 音视频同步示意图

参考时钟的选择也有多种方式，分别如下。

- 选取视频时间戳作为参考时钟源。
- 选取音频时间戳作为参考时钟源。
- 选取外部时间作为参考时钟源。

考虑到人对视频和音频的敏感度，在存在音频的情况下，应优先选择音频作为主时钟源。FFmpeg 在默认情况下也是使用音频作为参考时钟源的，处理同步的过程主要在解码后渲染前这个阶段。

8.4.6 音视频输出

回到 `ffplay.c` 中，如果发现待播放的文件中含有音频，那么在调用 `stream_component_open` 函数打开解码器时，其中也调用 `audio_open` 函数打开了音频输出设备。下面是 `audio_open` 函数的代码：

```
static int audio_open(void *opaque, int64_t wanted_channel_layout, int
wanted_nb_channels, int wanted_sample_rate, struct AudioParams *audio_hw_
params)
{
    SDL_AudioSpec wanted_spec, spec;
    const char *env;
    static const int next_nb_channels[] = {0, 0, 1, 6, 2, 6, 4, 6};
    static const int next_sample_rates[] = {0, 44100, 48000, 96000, 192000};
    int next_sample_rate_idx = FF_ARRAY_ELEMS(next_sample_rates) - 1;
    env = SDL_getenv("SDL_AUDIO_CHANNELS");
    //省略部分代码
    while (SDL_OpenAudio(&wanted_spec, &spec) < 0) {
        //省略部分代码
    }
    //省略部分代码
    return spec.size;
}
```

以上是音频部分，而视频部分渲染线程为 `video_refresh`，最后渲染图像的函数为 `video_image_display`，从 `video_audio_display` 判断 `VideoState` 是视频，调用 `video_image_display` 函数，代码如下：

```
static void video_display(VideoState *is)
{
    if (!screen)
        video_open(is, 0, NULL);
    if (is->audio_st && is->show_mode != SHOW_MODE_VIDEO)
```

```

        video_audio_display(is);
    else if (is->video_st)
        video_image_display(is);
}

```

video_image_display 定义如下:

```

static void video_image_display(VideoState *is)
{
    Frame *vp;
    Frame *sp;
    SDL_Rect rect;
    int i;
    vp = frame_queue_peek_last(&is->pictq);
    if (vp->bmp) {
        //省略部分代码
        calculate_display_rect(&rect, is->xleft, is->yp, is->width, is->
height, vp->width, vp->height, vp->sar);
        //省略部分代码
        SDL_DisplayYUVOverlay(vp->bmp, &rect);
    }
}

```

在最终的 SDL 中, 使用 OpenGL 来绘制图像。

第 9 章

直播技术

直播近年来越来越火，直播的形式也多种多样，如秀场直播、电竞直播、泛娱乐直播、校园直播，无论是哪种直播，都需要端和端之间的交互。本章将介绍直播技术中经常用到的相关知识点。

9.1 直播原理

通过计算机上的音视频输入设备或者手机端摄像头和麦克风实时录制的音视频流，编好码后通过直播协议将数据包实时发送给服务器端，服务器端通过流媒体协议把实时流分发出去，其他终端通过直播协议实时请求数据包，并进行解码播放。这就是直播的原理。

9.2 直播架构

直播架构主要分 3 块。第一块是采集数据推流过程，包括对数据流编码，通过流媒体协议传输到服务器上。第二块是服务器端收到推流数据后，进行内容分发及中间转存处理。最后一块是播放器进行拉流操作。这其中不只是播放音视频，还可以做一些实时美颜和滤镜效果。直播架构图如图 9-1 所示。

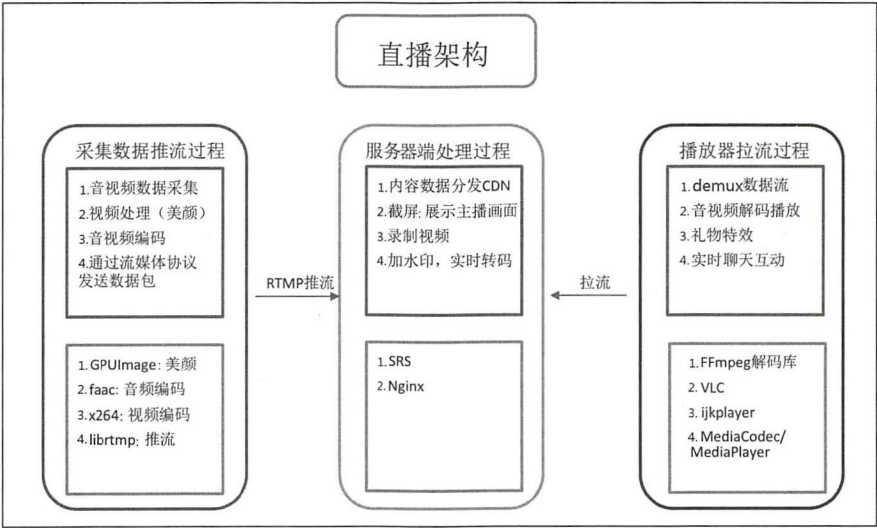


图 9-1 直播架构图

9.3 直播过程

直播过程主要涉及采集数据、渲染处理、编码数据、推流、CDN 分发、拉流、播放流数据等。

9.3.1 采集数据

采集数据时从不同平台的设备中获取原始音视频数据，用于美颜或编码处理。数据采集涉及音频采集和图像采集。

1. 采集内容

(1) 音频采集

音频数据既能与图像结合组成音视频数据，也支持单纯的音频数据输出（如电台）。音频的采集过程主要是通过设备设置采样率、采样数，将音频信号采集为 PCM 编码的原始数据，然后编码压缩成 MP3、AC3 等封装格式的数据分发出去。常见的音频封装格式有 MP3、AAC、OGG、WMA、Opus、FLAC、APE、M4A 和 AMR 等。声音的采样和量化如图 9-2 所示。

音频采集和编码面临的主要挑战在于去噪、回声消除（AEC）、静音检测（VAD）和各种音效处理等。

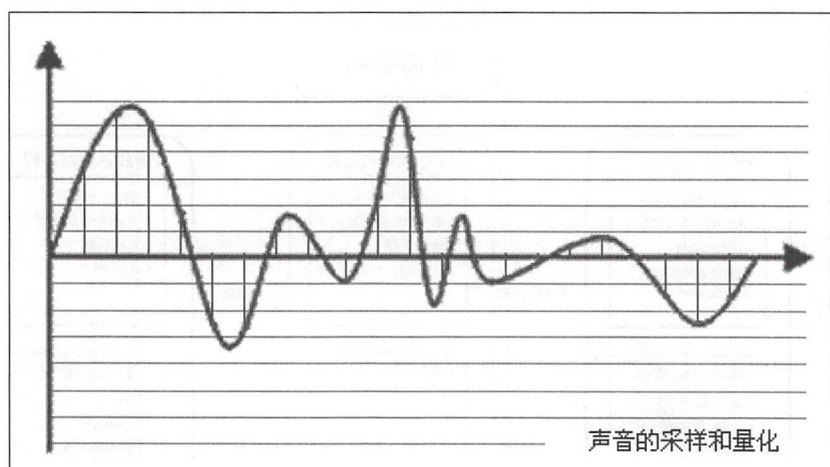


图 9-2 声音的采样和量化

在音频采集阶段，参考的主要技术参数如下。

- **采样率 (Samplerate)**: 采样率越高，数据量越大，音质越好。需要注意的是，并不是采样率越高，效果越好，不同声音效果的采样率有一定的阈值。
- **位宽**: 位宽表示一次能传递的数据宽度。就像公路的车道宽度有双向四车道、双向六车道等，车道越多一次能通过的汽车就越多一样，位宽越大，一次性能处理的数据就越多。音频采样过程中常用的位宽值有 8 位或者 16 位。
- **声道数 (Channel)**: 声道是指声音在录制或播放时在不同空间位置采集或回放的相互独立的音频信号，声道数是声音录制时的音源数量或回放时相应的扬声器数量。声道数为 1 和 2 分别被称为单声道和双声道，是比较常见的声道参数。

(2) 图像采集

图像采集就是通过摄像头或者可以采集图像的设备，获取一段时间内的图像内容。如手机摄像头采集的 NV21（一种 YUV 格式）格式数据，然后经过编码压缩成 H.264 等格式的数据，随后可以编码成不同的封装格式（如 FLV）传递或者直接通过流媒体协议（如 RTMP 协议）传递到服务器端。

在图像采集阶段，参考的主要技术参数如下。

- **图像格式**: 通常采用 YUV 格式存储原始数据信息，其中包含用 8 位表示的黑白图像灰度值，以及可由 RGB 共 3 种色彩组成的彩色图像。
- **传输通道**: 传输通道就是数据在传输时所利用的媒介，包括获取模块、数据类型模块及传输模块。如可以使用 TCP 传输或者 UDP 传输。

- **分辨率**：代表图像中存储的信息量，指每英寸图像内有多少个像素，图像分辨率的表达方式为“水平像素数×垂直像素数”。
- **采样频率（采样率）**：指每秒从连续信号中提取并组成离散信号的采样个数，如采样 720 像素视频，还是 540 像素视频，不同的采样个数对应的分辨率不一样，文件大小也不一样。
- **fps**：指画面每秒传输的帧数，通俗来讲就是动画或视频的画面数。直播一般设置为 15~20fps。

在实际开发中，常常要调整以上的一些参数，如 fps、分辨率，以获得不错的直播效果。

2. 采集源

（1）摄像头采集

对于视频内容，主要通过摄像头和专业摄像机进行采集。

（2）屏幕录制

屏幕录制，一般可以调用 Android 系统的 API 来捕捉屏幕进行录制。在一些音视频会议中，常常使用开源的桌面推流工具 OBS 进行屏幕录制和推流。OBS 屏幕录制和推流如图 9-3 所示。

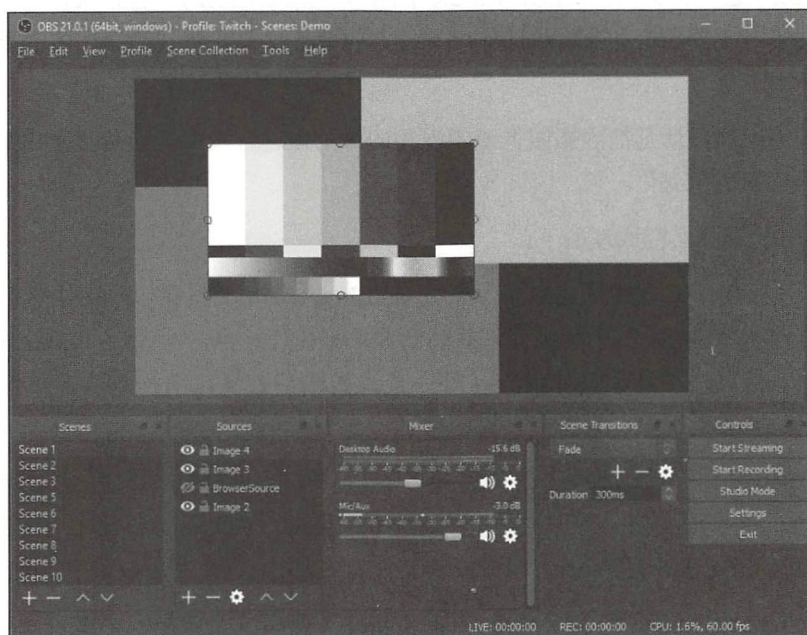


图 9-3 OBS 屏幕录制和推流

在图 9-3 中，我们把采集的内容分为图像和音频，其中图像的采集源包含摄像头、屏幕录制或者本地的视频文件，甚至其他需要重新定义和实现的采集源。而音频的采集源包含麦克风、系统声音或者本地音频文件，当然也可以为它定义别的采集源。

这样设计最大的好处在于，可以以轻量的设计方式支持丰富的采集源，而采集源的具体实现也可以交给使用者。

9.3.2 渲染处理

这里所说的渲染处理，主要是对从相机中采集来的数据进行二次处理。市面上比较好的美颜厂商有商汤、FaceUnity 等；而自己做美颜封装，可用的开源库主要是 GPUImage。

美颜的基本概念：通过一定的算法对原始图像数据进行二次处理并强化图像效果，不限于去掉不协调区域、边缘检测等。

GPU 工作原理：GPU 指图像运算工作的微处理器，GPU 主要利用显卡对图像的顶点坐标，通过图元组配，进行光栅化、顶点着色、片元着色等一系列管线操作。

OpenGL ES (Open Graphics Library for Embedded Systems, 开源嵌入式系统图形处理框架)：一套图形与硬件的接口，用于把处理好的图像显示到屏幕上。

GPUImage：是一个基于 OpenGL ES 2.0 图像和视频处理的跨平台框架，提供了各种各样的图像处理滤镜，支持相机和摄像机的实时滤镜，内置 120 多种滤镜效果，并且能够自定义图像滤镜。

滤镜处理的原理：就是把静态图像或者视频的每一帧进行图形变换后显示出来。它的本质是像素点的坐标和颜色变化。

GPUImage 处理画面的原理如下。

GPUImage 采用链式方式处理画面，通过 addTarget 函数为链条添加每个环节的对象，处理完一个 target，就会把上一个环节处理好的图像数据传递给下一个 target 去处理，这被称为 GPUImage 处理链。比如，墨镜原理，从外界传来光线，会经过墨镜过滤，再传给我们的眼睛，这样我们就能感受到大白天也会乌黑一片了。

一般的 target 可分为两类，一种是中间环节的 target，一般是各种滤镜，即 GPUImageFilter 或者其子类；另一种是最终环节的 target，GPUImageView，用于显示到屏幕上，或者是 GPUImageMovieWriter，写成视频文件。

GPUImage 处理主要分为 3 个环节，即 Source（视频、图像源）→filter（滤镜）→final target（处理后的视频、图像）。

GPUImage 的 Source 都继承自 GPUImageOutput 的子类, 作为 GPUImage 的数据源, 就好比外界的光线, 作为眼睛的输出源。Source 包括如下这几种。

- GPUImageVideoCamera: 用于实时拍摄视频。
- GPUImageStillCamera: 用于实时拍摄照片。
- GPUImagePicture: 用于处理已经拍摄好的图像, 比如 png、jpg 文件。
- GPUImageMovie: 用于处理已经拍摄好的视频, 比如 mp4 文件。

GPUImage 的 filter 一般是 GPUImageFilter 类或者其子类, 这个类继承自 GPUImageOutput, 并且遵守 GPUImageInput 协议, 这样既能流进又能流出。这好比墨镜, 光线经过墨镜的处理, 最终进入我们的眼睛。

GPUImage 的 final target 一般是 GPUImageView、GPUImageMovieWriter, 这好比我们的眼睛, 最终输入目标。

图 9-4 是一张美颜效果图。在美颜处理过程中最重要的是美颜算法, 如果读者想更多地研究美颜, 可以好好研究 GPUImage。

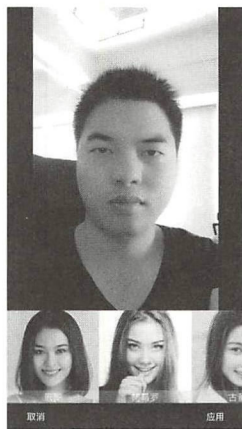


图 9-4 使用 GPUImage 的美颜效果图

9.3.3 编码数据

1. 视频编码的意义

视频编码指的是通过特定的压缩技术, 将某个视频格式文件转换成另一种视频格式文件的方式。这里要重点考虑的是压缩, 如当一个普通文件太大时, 可以用一些压缩工具压缩后传输, 以提高效率。视频压缩也是一样的, 采集到的视频数据一般较大, 经过 H.264 编码压缩后, 体积就会减小很多, 这样在传输过程中可以节省很多网络带宽资源。

2. 压缩原理

那为什么巨大的原始视频可以编码压缩成很小的视频呢? 这其中包含的技术有哪些? 其核心思想就是去除冗余信息, 如下所述。

(1) 空间冗余。在很多图像数据中, 像素间在行、列方向上都有很大的相关性, 相邻像素的值比较接近或者完全相同, 这种数据冗余叫作空间冗余。

(2) 时间冗余。在视频图像序列中, 相邻两帧图像数据有许多共同的地方, 这种共同性被称为时间冗余, 可采用运动补偿算法来去掉冗余信息。

(3) 视觉冗余。视觉冗余度是相对于人眼的视觉特性而言的, 人类视觉系统对图像的敏感

性是非均匀和非线性的，即人眼观察不到图像中的所有变化。

(4) 信息熵冗余。信息熵是指一组数据所携带的信息量，信息熵冗余指数据所携带的信息量少于数据本身，从而反映出数据冗余。

(5) 结构冗余。在有些图像的纹理区，图像的像素值存在着明显的分布模式。

(6) 知识冗余。对许多图像的理解与某些先验知识有相当大的相关性。这类规则的结构可由先验知识和背景知识得到，此类冗余被称为知识冗余。

3. 常用压缩编码方法分类

常用压缩编码方法分类如图 9-5 所示。

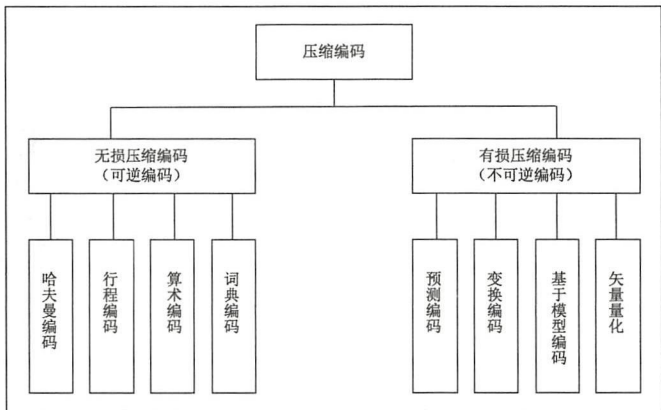


图 9-5 常用压缩编码方法分类

除了空间冗余和时间冗余的压缩，主要还有编码压缩和视觉压缩。下面是一个编码器主要的流程图。帧内编码过程如图 9-6 所示，帧间编码过程如图 9-7 所示。

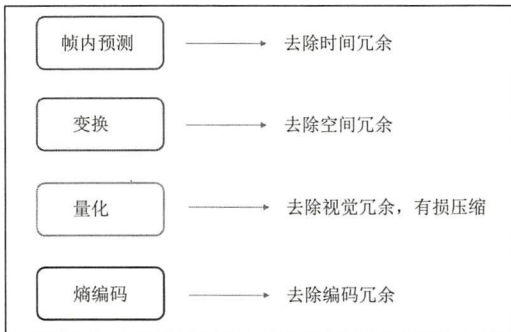


图 9-6 帧内编码过程

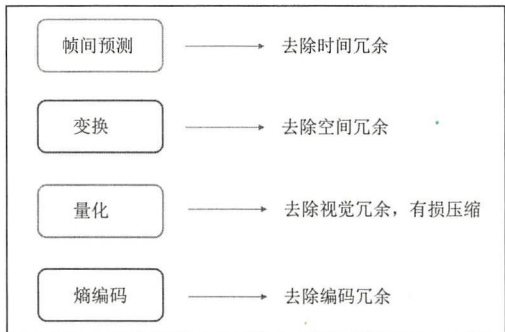


图 9-7 帧间编码过程

从图 9-6 和图 9-7 中可以看到两个过程的主要区别就是第一步不相同。其实这两个流程也是结合在一起的，我们通常说的 I 帧和 P 帧就分别采用了帧内编码和帧间编码。

- 预测：通过帧内预测和帧间预测降低视频图像的空间冗余和时间冗余。
- 变换：通过从时域到频域的变换，去除相邻数据之间的相关性，即去除空间冗余。
- 量化：通过用更粗糙的数据表示精细的数据来降低编码的数据量，或者通过去除人眼不敏感的信息来降低编码数据量。
- 熵编码：根据待编码数据的概率特性降低编码冗余。

9.3.4 推流

1. 推流协议

下面先介绍一下都有哪些推流协议，以及它们在直播领域的现状和优缺点。

- RTMP
- WebRTC
- 基于 UDP 的私有协议

(1) RTMP

RTMP 是 Real Time Messaging Protocol（实时消息传输协议）的首字母缩写。该协议基于 TCP，是一个协议族，包括 RTMP 基本协议及 RTMPT、RTMPS、RTMPE 等多个变种协议。RTMP 是一种被设计用来进行实时数据通信的网络协议，主要用在 Flash 平台和支持 RTMP 协议的流媒体/交互服务器之间进行音视频和数据通信。支持该协议的软件包括 Adobe Media Server、Ultrant Media Server、Red5 等。

RTMP 是目前主流的流媒体传输协议，广泛应用于直播领域，可以说市面上绝大多数的直播产品都采用了这个协议。

优点：

- 对 CDN 友好，主流的 CDN 厂商都支持该协议。
- 协议简单，在各平台上实现容易。

缺点：

- 基于 TCP，传输成本高，在弱网环境丢包率高的情况下问题明显。
- 不支持浏览器推送。
- Adobe 私有协议，Adobe 已经不再更新该协议。

(2) WebRTC

WebRTC，名称源自 Web Real Time Communication（网页即时通信），是一个支持 Web 浏览器进行实时语音对话或视频对话的 API。它于 2011 年 6 月 1 日开源，并在 Google、Mozilla、Opera 的支持下被纳入万维网联盟的 W3C 推荐标准。

目前其主要应用于视频会议和连麦中。WebRTC 协议分层图如图 9-8 所示。

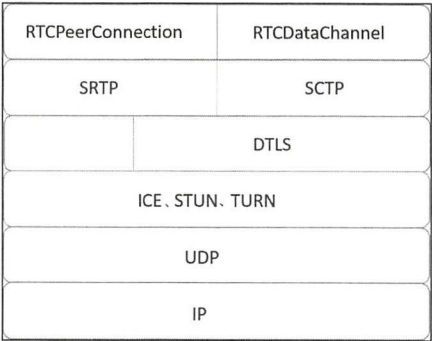


图 9-8 WebRTC 协议分层图

优点：

- W3C 标准，主流浏览器支持程度高。
- Google 在背后支撑，并针对各平台有参考实现。
- 底层基于 SRTP 和 UDP，在弱网情况下优化空间大。
- 可以实现点对点通信，通信双方延时低。

缺点：

ICE、STUN、TURN 等传统 CDN 没有提供类似的服务。

(3) 基于 UDP 的私有协议

有些直播应用会使用 UDP 作为底层协议开发自己的私有协议。由于 UDP 在弱网环境下的优势，因此通过一些定制化的调优可以达到比较好的弱网优化效果；但同样因为其是私有协议，故势必会有现实问题。

优点：

有更多的空间进行定制化优化。

缺点：

- 开发成本高。
- 对 CDN 不友好，需要自建 CDN 或者和 CDN 达成协议。
- 独立作战，无法和社区一起演进。

2. 推流过程

推流过程就是把编码后的数据打包并通过直播协议发送给流媒体服务器的过程，如图 9-9 所示。

- 经过输出设备得到原始的采样数据——视频数据（YUV）和音频数据（PCM）。

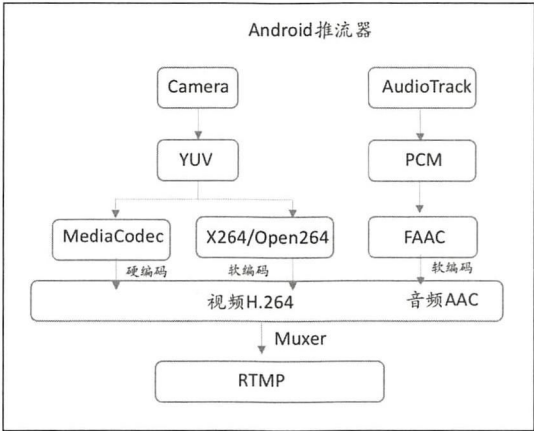


图 9-9 Android 推流器过程

- 使用硬编码（MediaCodec）或软编码（FFmpeg）来编码压缩音视频数据。
- 分别得到已编码的 H.264 视频数据和 AAC 音频数据。
- 将已编码的音视频数据封装成不同的视频封装格式的数据文件（如 FLV、TS、MPEG-TS）。
- 使用 HLS 协议的时候加上这一步（HLS 分段生成策略及 m3u8 索引文件）。
- 通过流媒体协议上传到服务器。
- 服务器通过相关协议对内容进行分发。

3. 推流优化

在推流过程中经常会遭遇网络不好、断流的情况，所以需要一定的策略。在推流端 ping 视频中心地址，测试是否有丢包现象。同时在网络抖动时，需要动态调整一些参数以便推流。在网络断后重连时，需要重新优先发送音频数据，保证用户能听到声音。视频数据随后到达，使观众看到画面变化，并逐步回归到音视频同步。

9.3.5 CDN 分发

1. CDN 技术原理

CDN 的全称为 Content Delivery Network，即内容分发网络，是一个策略性部署的整体系统，主要用来解决由于网络带宽小、用户访问量大、网点分布不均匀等造成的用户访问网站速度慢的问题。这中间会有很多 CDN 节点，简单一点理解就相当于让计算机选择最优网络。具体实现是，通过在现有的网络中增加一层新的网络架构，将网站内容发布到离用户最近的网络节点上。这样用户可以就近获取所需的内容，解决之前的网络拥塞、访问延时高的问题，提高用户体验，如图 9-10 所示。

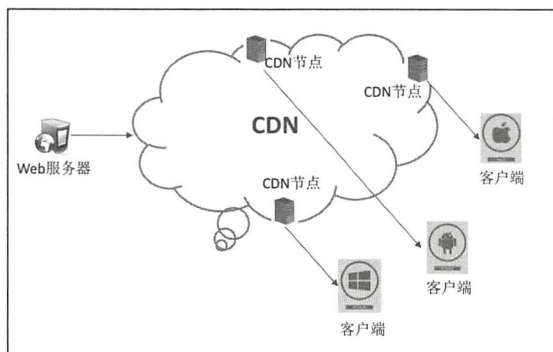


图 9-10 CDN 分发过程

在图 9-11 中，对不同流媒体所使用的节点和协议做了区分，使得网络拥塞和访问延时减少，带宽得到良好的控制等。在 CDN 直播中常用的流媒体协议包括 RTMP、HLS、HTTP-FLV 等。

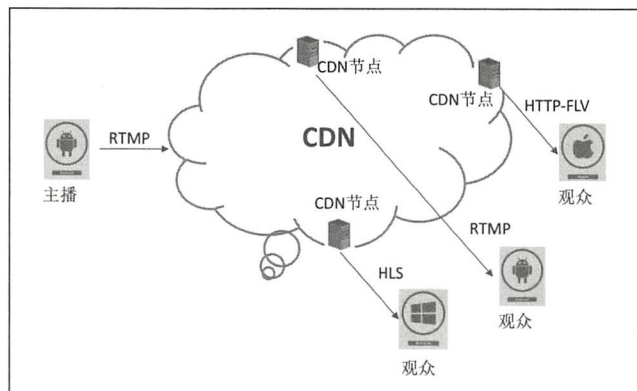


图 9-11 直播推流到 CDN 后，不同协议的分发过程

RTMP (Real Time Messaging Protocol) 基于 TCP 协议, 是由 Adobe 公司为 Flash 播放器和服务器之间进行音视频传输开发的开放协议。HLS (HTTP Live Streaming) 基于 HTTP 协议, 是 Apple 公司开放的音视频传输协议。HTTP-FLV 则将 RTMP 封装在 HTTP 协议之上, 可以更好地穿透防火墙等。

2. CDN 的常用架构

CDN 架构设计比较复杂, 而且不同的 CDN 厂商也在对其架构进行不断的优化, 所以对这些架构不能统一而论。这里只对一些基本的架构进行简单的剖析。

CDN 主要包含源站、缓存服务器、智能 DNS、客户端等几个主要组成部分。

- **源站:** 是指发布内容的原始站点。添加、删除和更改网站的文件都是在源站上进行的, 另外缓存服务器所抓取的对象也全部来自源站。对于直播来说, 源站为主播客户端。
- **缓存服务器:** 是直接提供给用户访问的站点资源, 由一台或数台服务器组成。当用户发起访问时, 其访问请求被智能 DNS 定位到离他较近的缓存服务器。如果用户所请求的内容刚好在缓存里面, 则直接把内容返还给用户; 如果访问所需的内容没有被缓存, 则缓存服务器向邻近的缓存服务器或直接向源站抓取内容, 然后返还给用户。
- **智能 DNS:** 是整个 CDN 技术的核心, 它主要根据用户的来源, 以及当前缓存服务器的负载情况等, 将其访问请求指向离用户比较近且负载较小的缓存服务器。通过智能 DNS 解析, 让用户访问同服务商、负载较小的服务器, 可以缓解网络访问慢的情况, 达到加速的作用。
- **客户端:** 即发起访问的普通用户。对于直播来说, 就是观众 (客户端), 例如手机客户端、PC 客户端。

CDN 常用架构如图 9-12 所示。

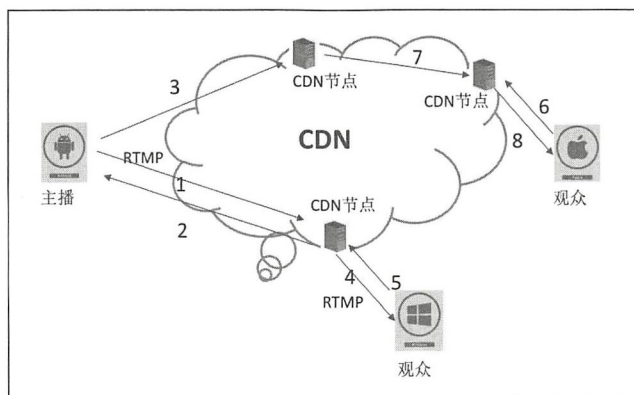


图 9-12 CDN 常用架构

整个流程描述如下。

主播开始进行直播，向智能 DNS 发送解析请求；智能 DNS 返回最优 CDN 节点的 IP 地址；主播端采集音视频数据，发送给 CDN 节点，CDN 节点进行缓存等处理；观众要观看这个主播的视频，向智能 DNS 发送解析请求；智能 DNS 返回最优 CDN 节点的 IP 地址；观众向 CDN 节点请求音视频数据；CDN 节点同步其他节点的音视频数据；CDN 节点将音视频数据发送给观众。

3. CDN 的缺点

大概了解了 CDN 的技术原理后，我们在做直播选型时，还需要了解一个方案的优缺点。接下来，分析一下 CDN 的短板。

总的来说，CDN 主要有如下缺点。

(1) 播放延时（网络延时）

网络延时这里指的是从主播端采集到观众播放之间的时间差。这里不考虑主播端采集对视频进行编码的时间，以及观众端播放对视频进行解码的时间，仅考虑网络传输中的延时。例如，图 9-13 中的网络延时过程。

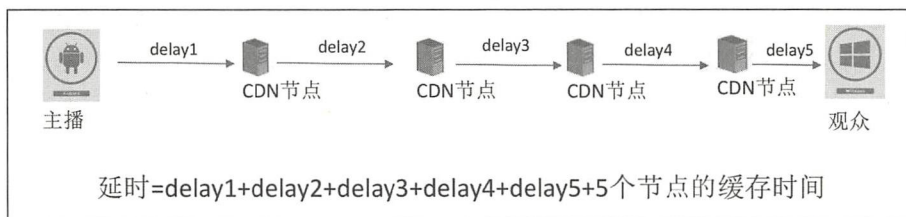


图 9-13 网络延时过程

(2) 网络抖动

网络抖动，是指数据包的到达顺序、间隔等与发出时不一致。比如，发送 100 个数据包，每个包间隔 1s 发出。结果第 30 个包在传输过程中遇到网络拥塞，造成包 30 不是紧跟着包 29 到达的，而是延迟到包 90 后面才到达。在直播中，这种抖动的效果实际上跟丢包是一样的。因为你不能依照接收顺序把内容播放出来，所以这样会造成失真。网络抖动会造成播放延时相应地增大。如果网络中抖动得较大，会造成播放卡顿等现象。

(3) 网络丢包

CDN 直播中用到的 RTMP、HLS、HTTP-FLV 等协议都建立在 TCP 协议的基础之上。TCP 的一个很重要的特性是可靠性，即不会发生数据丢失的问题。为了保证可靠性，TCP 在传输过程中有三次握手。首先客户端会向服务器端发送连接请求，服务器端同意后，客户端会确认这

次连接，这就是三次握手。接着，客户端开始发送数据，每次发送一批数据，得到服务器端的“收到”确认后，继续发送下一批。TCP 协议为了保证数据传到目的地，会有自动重传机制。如果传输中发生了丢包，没有收到对方端发出的“收到”信号，那么就会自动重传丢失的包，一直到超时。

由于互联网的网络状况是变化着的，加之主播端的网络状况是无法控制的，因此当网络中的丢包率开始升高时，重传会导致延时不断增大，甚至导致不断尝试重连等情况，这样不能有效缓存，在严重的情况下甚至会导致观众端无法观看视频。

9.3.6 拉流

根据协议类型（如 RTMP、RTP、RTSP、HTTP 等），与服务器建立连接并接收数据。

（1）解析二进制数据，从中找到相关流信息。

（2）根据不同的封装格式（如 FLV、TS）解复用。

（3）分别得到已编码的 H.264 视频数据和 AAC 音频数据。

（4）使用硬解码（MediaCodec）或软解码（FFmpeg）来解压音视频数据。

（5）经过解码后得到原始的视频数据（YUV）和音频数据（PCM）。

（6）因为音频和视频解码是分开的，所以在解码后，需要做音视频同步。通常有一些音视频同步算法。如在 FFmpeg 中，以音频作为参考时钟，视频用于比较当前时钟和音频时钟的差值，如果快了，就需要增大延时，以便下一帧显示得晚些；如果慢了，就需要减少延时，加快显示下一帧。

（7）最后把同步的音频数据发送到耳机或外放设备上播放，将视频数据发送到屏幕上显示。

了解了播放器的播放流程后，可以优化首帧显示时间。从第（2）点入手，通过预设解码器类型，省去探测文件类型的时间。从第（5）点入手，缩小视频数据探测范围，同时也意味着减少了需要下载的数据量，特别是在网络不好的时候，减少下载的数据量能为启动播放节省大量的时间，在检测到 I 帧数据后就立马返回并进入播放解码环节。

9.3.7 播放流数据

播放流数据，一般涉及几个过程。首先进行 access 操作，也就是获取数据流；然后进行 demux 操作，也就是解复用，将数据流分离成音频流和视频流；接着将音频流送入音频解码器，将视频流送入视频解码器；最后进行音视频同步并输出。通常直接使用第三方播放器，仅仅了解 API 调用就行，其他的则交由播放内核去做。如在 Android 上使用 MediaPlayer，仅仅需要图 9-14 中的几步就可以进行播放了。

```
public void startPlayUri(Uri uri) {
    mMediaPlayer = new MediaPlayer();
    try {
        mMediaPlayer.setDataSource(mContext, uri);
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (SecurityException e) {
        e.printStackTrace();
    } catch (IllegalStateException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }

    mMediaPlayer.setOnPreparedListener(mPreparedListener);
    mMediaPlayer.setOnVideoSizeChangedListener(mVideoSizeChangeListener);
    mMediaPlayer.setOnErrorListener(mOnErrorListener);
    mMediaPlayer.setOnCompletionListener(mOnCompletionListener);
    mMediaPlayer.prepareAsync();
}
```

图 9-14 MediaPlayer 常规使用方式

基本上第三方播放器（如 VLC、IjkPlayer）对外提供的接口都是从 Android 的 MediaPlayer 接口扩展来的。对于应用来说，只要把 URL 发送给播放器并调用 PrepareAsync 函数，然后等待 onPrepared 回调成功后，就可以调用 start 函数起播了。

下面介绍软硬解码的选择。

考虑到 Android 平台的各种兼容性问题，需要根据不同手机的解码能力来进行硬解码和软解码的选择。这里也介绍一些经验，但根本问题是，没有一个通用方案能最优适配所有操作系统和机型。

- 硬解码：推荐 Andorid 4.1.2（API 16）或以上版本使用硬解码，而 4.1.2 以下版本使用软解码。
- 软解码：主要利用 CPU 执行大量运算来解码。虽然这样牺牲了功耗，但是在部分细节方面表现较优，且可控性强，兼容性也强，出错情况少，在硬解码无能为力的情况下，软解码也不失为一种不错的选择。

下面是软硬解码的优缺点对比，如表 9-1 所示。

表 9-1 软硬解码的优缺点对比

	优 点	缺 点
软解码	1. 兼容性强，对系统版本要求低，出错情况少； 2. 解码方面，软解码的色彩一般比硬解码的柔和； 3. 编码的可操作空间比较大，自由度高	1. CPU 消耗较大； 2. 机器容易发热； 3. 功耗较高
硬解码	功耗低、执行效率低	1. 不同型号的芯片对编解码的实现不同，并不能保证编解码效果与其他机型一样或不出错； 2. 可控性差，依赖底层编解码实现

9.3.8 直播推流完整案例

整个直播推流分成 Java 层、JNI 层，以及 C/C++ 层。Java 层主要负责传入一些推流服务器地址以及由相机采集的数据。MainActivity.java 的代码如下：

```
package com.hejunlin.ffmpegpshcamerastream;

import android.app.Activity;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.SurfaceView;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import com.hejunlin.ffmpegpshcamerastream.core.AVStreamPush;

public class MainActivity extends Activity {

    //private static final String URL = "rtmp://192.168.0.101:1935/test/
live";
    private static final String URL = "rtmp://202.197.224.44:1935/
yuiop2/live";
    private AVStreamPush mAVStreamPush;
    private Button mStartLive;
    private Button mChangePreView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mStartLive = (Button) findViewById(R.id.bt_start_live);
        mChangePreView = (Button) findViewById(R.id.bt_change_camera);
        SurfaceView surfaceView = (SurfaceView) findViewById(R.id.surface);
        //相机图像的预览
        mAVStreamPush = new AVStreamPush(surfaceView.getHolder());
        mStartLive.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Button btn = (Button) view;
                if (btn.getText().equals("开始直播")) {
                    mAVStreamPush.startPush(URL);
                    btn.setText("停止直播");
                } else {
                    mAVStreamPush.stopPush();
                }
            }
        });
    }
}
```



```

        btn.setText("开始直播");
    }
}

});

mChangePreView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        mAVStreamPush.switchCamera();
    }
});
}
}

```

NativePlayer.java 文件的作用是定义 native 函数，JNI 层有对应的实现方法，代码如下：

```

package com.hejunlin.ffmpegpshcamerastream;

public class NativePlayer {

    public native void startPush(String url);

    public native void stopPush();

    public native void release();

    /**
     * 设置视频参数
     * @param width
     * @param height
     * @param bitrate
     * @param fps
     */
    public native void setVideoOptions(int width, int height, int bitrate,
int fps);

    /**
     * 设置音频参数
     * @param sampleRateInHz
     * @param channel
     */
    public native void setAudioOptions(int sampleRateInHz, int channel);
}

```

```

/**
 * 发送视频数据
 * @param data
 */
public native void sendVideoPacket(byte[] data);

/**
 * 发送音频数据
 * @param data
 * @param len
 */
public native void sendAudioPacket(byte[] data, int len);

static{
    System.loadLibrary("yuiopLiveFFmpeg");
}
}

```

BasePush.java 是一个基类，定义推拉的简单接口，代码如下：

```

package com.hejunlin.ffmpegpshcamerastream.core;

public abstract class BasePush {

    public abstract void startPush();

    public abstract void stopPush();

    public abstract void release();

}

```

VideoPush.java 类的主要作用是采集相机中的视频数据，然后通过预览回调，把采集到的数据回传到 JNI 层中进行美颜或是编码，代码如下：

```

package com.hejunlin.ffmpegpshcamerastream.core;

import java.io.IOException;
import java.lang.reflect.Method;

import com.hejunlin.ffmpegpshcamerastream.NativePlayer;
import com.hejunlin.ffmpegpshcamerastream.model.VideoInfo;

import android.graphics.ImageFormat;
import android.graphics.PixelFormat;

```



```

import android.hardware.Camera;
import android.hardware.Camera.CameraInfo;
import android.hardware.Camera.PreviewCallback;
import android.os.Build;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceHolder.Callback;

public class VideoPush extends BasePush implements Callback, PreviewCallback{

    private static final String TAG = VideoPush.class.getSimpleName();
    private SurfaceHolder mSurfaceHolder;
    private Camera mCamera;
    private VideoInfo mVideoInfo;
    private byte[] buffers;
    private boolean isPushing = false;
    private NativePlayer nativePlayer;

    public VideoPush(SurfaceHolder surfaceHolder, VideoInfo videoInfo,
NativePlayer nativePlayer) {
        this.mSurfaceHolder = surfaceHolder;
        this.mVideoInfo = videoInfo;
        this.nativePlayer = nativePlayer;
        surfaceHolder.addCallback(this);

        surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }

    @Override
    public void startPush() {
        //设置视频参数
        nativePlayer.setVideoOptions(mVideoInfo.getWidth(),
mVideoInfo.getHeight(), mVideoInfo.getBitrate(), mVideoInfo.getFps());
        isPushing = true;
    }

    @Override
    public void stopPush() {
        isPushing = false;
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        startPreview();
    }

```



```

    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width,
int height) {
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {

    }

    @Override
    public void release() {
        stopPreview();
    }

    /**
     * 切换摄像头
     */
    public void switchCamera() {
        if(mVideoInfo.getCameraId() == CameraInfo.CAMERA_FACING_BACK){
            mVideoInfo.setCameraId(CameraInfo.CAMERA_FACING_FRONT);
        }else{
            mVideoInfo.setCameraId(CameraInfo.CAMERA_FACING_BACK);
        }
        //重新预览
        stopPreview();
        startPreview();
    }

    /**
     * 开始预览
     */
    private void startPreview() {
        try {
            //SurfaceView 初始化完成, 开始相机预览
            mCamera = Camera.open(mVideoInfo.getCameraId());
            Camera.Parameters parameters = mCamera.getParameters();
            //设置相机参数
            parameters.setPreviewFormat(ImageFormat.NV21);
            //YUV 预览图像的像素格式
            parameters.setPictureSize(720,1080);
            parameters.setFlashMode(Camera.Parameters.FLASH_MODE_TORCH);

```

```

        parameters.setFocusMode(Camera.Parameters.FOCUS_MODE_CONTINUOUS_PICTURE);
        parameters.setPreviewSize(mVideoInfo.getWidth(), mVideoInfo.getHeight());
        //预览画面宽高值
        setDispaly(parameters, mCamera);
        mCamera.setParameters(parameters);
        mCamera.setPreviewDisplay(mSurfaceHolder);
        //获取预览图像数据
        buffers = new byte[mVideoInfo.getWidth() * mVideoInfo.getHeight() * 4];
        mCamera.addCallbackBuffer(buffers);
        mCamera.setPreviewCallbackWithBuffer(this);
        mCamera.startPreview();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private void setDispaly(Camera.Parameters parameters, Camera camera) {
    if (Integer.parseInt(Build.VERSION.SDK) >= 8) {
        setDisplayOrientation(camera, 90);
    } else {
        parameters.setRotation(90);
    }
}

private void setDisplayOrientation(Camera camera, int i) {
    Method downPolymorphic;
    try {
        downPolymorphic = camera.getClass().getMethod("setDisplayOrientation", new Class[]{int.class});
        if (downPolymorphic != null) {
            downPolymorphic.invoke(camera, new Object[]{i});
        }
    } catch (Exception e) {
    }
}

/**
 * 停止预览
 */

private void stopPreview() {

```



```

        if(mCamera != null){
            mCamera.stopPreview();
            mCamera.release();
            mCamera = null;
        }
    }

    @Override
    public void onPreviewFrame(byte[] data, Camera camera) {
        if(mCamera != null){
            mCamera.addCallbackBuffer(buffer);
        }

        if(isPushing){
            //在回调函数中获取图像数据，然后给 Native 代码编码
            nativePlayer.sendVideoPacket(data);
        }
    }
}

```

AudioPush.java 类的主要作用是采集音频数据，并回传到 JNI 层进行编码处理，代码如下：

```

package com.hejunlin.ffmpegpshcamerastream.core;

import android.media.AudioFormat;
import android.media.AudioRecord;
import android.media.MediaRecorder.AudioSource;

import com.hejunlin.ffmpegpshcamerastream.NativePlayer;
import com.hejunlin.ffmpegpshcamerastream.model.AudioInfo;

public class AudioPush extends BasePush {

    private AudioInfo mAudioInfo;
    private AudioRecord audioRecord;
    private boolean isPushing = false;
    private int minBufferSize;
    private NativePlayer nativePlayer;

    public AudioPush(AudioInfo audioInfo, NativePlayer nativePlayer) {
        this.mAudioInfo = audioInfo;
        this.nativePlayer = nativePlayer;

        int channelConfig = mAudioInfo.getChannel() == 1 ?

```



```

        AudioFormat.CHANNEL_IN_MONO : AudioFormat.CHANNEL_IN_STEREO;
        //最小缓冲区大小
        minBufferSize = AudioRecord.getMinBufferSize(mAudioInfo.getSample
RateInHz(), channelConfig, AudioFormat.ENCODING_PCM_16BIT);
        audioRecord = new AudioRecord(AudioSource.MIC,
            mAudioInfo.getSampleRateInHz(),
            channelConfig,
            AudioFormat.ENCODING_PCM_16BIT, minBufferSize);
    }

    @Override
    public void startPush() {
        isPushing = true;
        //启动一个录音子线程
        new Thread(new AudioRecordTask()).start();
    }

    @Override
    public void stopPush() {
        isPushing = false;
        audioRecord.stop();
    }

    @Override
    public void release() {
        if(audioRecord != null){
            audioRecord.release();
            audioRecord = null;
        }
    }

    class AudioRecordTask implements Runnable{

        @Override
        public void run() {
            //开始录音
            audioRecord.startRecording();
            while(isPushing){
                //通过 AudioRecord 不断读取音频数据
                byte[] buffer = new byte[minBufferSize];
                int len = audioRecord.read(buffer, 0, buffer.length);
                if(len > 0){
                    //传给 Native 代码, 进行音频编码
                    nativePlayer.sendAudioPacket(buffer, len);
                }
            }
        }
    }

```

```

    }
}
}
}
}

```

AVStreamPush.java 类的主要作用是实例化音视频推流器，并设置一些参数，提供推流的接口和回调函数。其代码如下：

```

package com.hejunlin.ffmpegpshcamerastream.core;

import com.hejunlin.ffmpegpshcamerastream.NativePlayer;
import com.hejunlin.ffmpegpshcamerastream.model.AudioInfo;
import com.hejunlin.ffmpegpshcamerastream.model.VideoInfo;

import android.hardware.Camera.CameraInfo;
import android.view.SurfaceHolder;
import android.view.SurfaceHolder.Callback;

public class AVStreamPush implements Callback {

    private SurfaceHolder surfaceHolder;
    private VideoPush videoPush;
    private AudioPush audioPush;
    private NativePlayer nativePlayer;

    public AVStreamPush(SurfaceHolder surfaceHolder) {
        this.surfaceHolder = surfaceHolder;
        surfaceHolder.addCallback(this);
        prepare();
    }

    /**
     * 预览准备
     */
    private void prepare() {
        nativePlayer = new NativePlayer();

        //实例化视频推流器
        VideoInfo videoParam = new VideoInfo(480, 320, CameraInfo.CAMERA_
FACING_BACK);
        videoPush = new VideoPush(surfaceHolder, videoParam, nativePlayer);

        //实例化音频推流器

```



```

        AudioInfo audioInfo = new AudioInfo();
        audioPush = new AudioPush(audioInfo, nativePlayer);
    }

    /**
     * 切换摄像头
     */
    public void switchCamera() {
        videoPush.switchCamera();
    }

    /**
     * 开始推流
     */
    public void startPush(String url) {
        videoPush.startPush();
        audioPush.startPush();
        nativePlayer.startPush(url);
    }

    /**
     * 停止推流
     */
    public void stopPush() {
        videoPush.stopPush();
        audioPush.stopPush();
        nativePlayer.stopPush();
    }

    /**
     * 释放资源
     */
    private void release() {
        videoPush.release();
        audioPush.release();
        nativePlayer.release();
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder) {

    }

    @Override

```



```

    public void surfaceChanged(SurfaceHolder holder, int format, int width,
int height) {

    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        stopPush();
        release();
    }
}

```

下面介绍 Model 的相关代码。

AudioInfo.java 类是音频推流实体对象，代码如下：

```

package com.hejunlin.ffmpegpshcamerastream.model;

public class AudioInfo {

    //采样率
    private int sampleRateInHz = 44100;
    //声道个数
    private int channel = 1;

    public AudioInfo() {
    }

    public AudioInfo(int sampleRateInHz, int channel) {
        super();
        this.sampleRateInHz = sampleRateInHz;
        this.channel = channel;
    }

    public int getSampleRateInHz() {
        return sampleRateInHz;
    }

    public void setSampleRateInHz(int sampleRateInHz) {
        this.sampleRateInHz = sampleRateInHz;
    }

    public int getChannel() {
        return channel;
    }
}

```

```

    }

    public void setChannel(int channel) {
        this.channel = channel;
    }

}

```

VideoInfo.java 类是视频推流实体对象，代码如下：

```

package com.hejunlin.ffmpegpshcamerastream.model;

/**
 * 视频数据参数
 */
public class VideoInfo {

    private int width;
    private int height;
    //码率为 480kb/s
    private int bitrate = 480000;
    //帧频默认为 25fps
    private int fps = 25;
    private int cameraId;

    public VideoInfo(int width, int height, int cameraId) {
        super();
        this.width = width;
        this.height = height;
        this.cameraId = cameraId;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {

```

```

        this.height = height;
    }

    public int getCameraId() {
        return cameraId;
    }

    public void setCameraId(int cameraId) {
        this.cameraId = cameraId;
    }

    public int getBitrate() {
        return bitrate;
    }

    public void setBitrate(int bitrate) {
        this.bitrate = bitrate;
    }

    public int getFps() {
        return fps;
    }

    public void setFps(int fps) {
        this.fps = fps;
    }
}

```

com_hejunlin_ffmpegpushcamerastream_NativePlayer.h 是通过 java -h 命令生成的 NativePlayer 对应的头文件。实现 NativePlayer 中的 native 函数需要引入此头文件，其代码如下：

```

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_hejunlin_ffmpegpushcamerastream_NativePlayer */

#ifndef _Included_com_hejunlin_ffmpegpushcamerastream_NativePlayer
#define _Included_com_hejunlin_ffmpegpushcamerastream_NativePlayer
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      com_hejunlin_ffmpegpushcamerastream_NativePlayer
 * Method:     startPush

```



```

    * Signature: (Ljava/lang/String;)V
    */
    JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
startPush
        (JNIEnv *, jobject, jstring);

    /*
    * Class:      com_hejunlin_ffmpegpushcamerastream_NativePlayer
    * Method:      stopPush
    * Signature:   ()V
    */
    JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
stopPush
        (JNIEnv *, jobject);

    /*
    * Class:      com_hejunlin_ffmpegpushcamerastream_NativePlayer
    * Method:      release
    * Signature:   ()V
    */
    JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
release
        (JNIEnv *, jobject);

    /*
    * Class:      com_hejunlin_ffmpegpushcamerastream_NativePlayer
    * Method:      setVideoOptions
    * Signature:   (III)V
    */
    JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
setVideoOptions
        (JNIEnv *, jobject, jint, jint, jint, jint);

    /*
    * Class:      com_hejunlin_ffmpegpushcamerastream_NativePlayer
    * Method:      setAudioOptions
    * Signature:   (II)V
    */
    JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
setAudioOptions
        (JNIEnv *, jobject, jint, jint);

    /*
    * Class:      com_hejunlin_ffmpegpushcamerastream_NativePlayer

```

```

    * Method:      sendVideoPacket
    * Signature:   ([B]V
    */
JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
sendVideoPacket
    (JNIEnv *, jobject, jbyteArray);

/*
 * Class:      com_hejunlin_ffmpegpushcamerastream_NativePlayer
 * Method:      sendAudioPacket
 * Signature:   ([BI]V
 */
JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
sendAudioPacket
    (JNIEnv *, jobject, jbyteArray, jint);

#ifdef __cplusplus
}
#endif
#endif

```

DoubleQueue.h 是双端队列中的头文件，代码如下：

```

#ifndef _QUEUE_H
#define _QUEUE_H
extern int create_queue();
extern int destroy_queue();
extern int queue_is_empty();
extern int queue_size();
extern void* queue_get(int index);
extern void* queue_get_first();
extern void* queue_get_last();
extern int queue_insert(int index, void *pval);
extern int queue_insert_first(void *pval);
extern int queue_append_last(void *pval);
extern int queue_delete(int index);
extern int queue_delete_first();
extern int queue_delete_last();
#endif//_QUEUE_H

```

DoubleQueue.c，即双端队列，主要用于编码数据的存放和消耗，代码如下：

```

#include <stdio.h>
#include <malloc.h>

typedef struct queue_node {

```



```

    struct queue_node* prev;
    struct queue_node* next;
    void *p; //节点的值
} node;

//表头。注意，表头不存放元素值!!!
static node *phead = NULL;
static int count = 0;

static node* create_node(void *pval) {
    node *pnode = NULL;
    pnode = (node *) malloc(sizeof(node));
    if (pnode) {
        //默认的，pnode 的前一节点和后一节点都指向它自身
        pnode->prev = pnode->next = pnode;
        //节点的值为 pval
        pnode->p = pval;
    }
    return pnode;
}

//新建“双向链表”。成功，返回 0；否则，返回-1
int create_queue() {
    phead = create_node(NULL);
    if (!phead) {
        return -1;
    }
    //设置“节点个数”为 0
    count = 0;
    return 0;
}

//“双向链表是否为空”
int queue_is_empty() {
    return count == 0;
}

//返回“双向链表的大小”
int queue_size() {
    return count;
}

//获取“双向链表中第 index 位置的节点”
static node* get_node(int index) {
    if (index < 0 || index >= count) {

```



```

        return NULL;
    }
    if (index <= (count / 2)) {
        int i = 0;
        node *pnode = phead->next;
        while ((i++) < index)
            pnode = pnode->next;
        return pnode;
    }
    int j = 0;
    int rindex = count - index - 1;
    node *rnode = phead->prev;
    while ((j++) < rindex)
        rnode = rnode->prev;
    return rnode;
}

//获取“第一个节点”
static node* get_first_node() {
    return get_node(0);
}

//获取“最后一个节点”
static node* get_last_node() {
    return get_node(count - 1);
}

//获取“双向链表中第 index 位置的元素”。成功，返回节点值；否则，返回-1。
void* queue_get(int index) {
    node *pindex = get_node(index);
    if (!pindex) {
        return NULL;
    }
    return pindex->p;
}

//获取“双向链表中第 1 个元素的值”
void* queue_get_first() {
    return queue_get(0);
}

void* queue_get_last() {
    return queue_get(count - 1);
}

//将“pval”插入 index 位置。成功，返回 0；否则，返回-1

```



```
int queue_insert(int index, void* pval) {
    //插入表头
    if (index == 0)
        return queue_insert_first(pval);
    //获取要插入的位置对应的节点
    node *pindex = get_node(index);
    if (!pindex)
        return -1;
    //创建“节点”
    node *pnode = create_node(pval);
    if (!pnode)
        return -1;
    pnode->prev = pindex->prev;
    pnode->next = pindex;
    pindex->prev->next = pnode;
    pindex->prev = pnode;
    //节点个数加1
    count++;
    return 0;
}
```

//将“pval”插入表头位置

```
int queue_insert_first(void *pval) {
    node *pnode = create_node(pval);
    if (!pnode)
        return -1;
    pnode->prev = phead;
    pnode->next = phead->next;
    phead->next->prev = pnode;
    phead->next = pnode;
    count++;
    return 0;
}
```

//将“pval”插入末尾位置

```
int queue_append_last(void *pval) {
    node *pnode = create_node(pval);
    if (!pnode)
        return -1;
    pnode->next = phead;
    pnode->prev = phead->prev;
    phead->prev->next = pnode;
    phead->prev = pnode;
    count++;
    return 0;
}
```




```
}
//删除“双向链表中第 index 位置的节点”。成功，返回 0；否则，返回-1
int queue_delete(int index) {
    node *pindex = get_node(index);
    if (!pindex) {
        return -1;
    }
    pindex->next->prev = pindex->prev;
    pindex->prev->next = pindex->next;
    free(pindex);
    count--;
    return 0;
}
//删除第一个节点
int queue_delete_first() {
    return queue_delete(0);
}
//删除最后一个节点
int queue_delete_last() {
    return queue_delete(count - 1);
}
//撤销“双向链表”。成功，返回 0；否则，返回-1
int destroy_queue() {
    if (!phead) {
        return -1;
    }
    node *pnode = phead->next;
    node *ptmp = NULL;
    while (pnode != phead) {
        ptmp = pnode;
        pnode = pnode->next;
        free(ptmp);
    }
    free(phead);
    phead = NULL;
    count = 0;
    return 0;
}
```

Log.h 头文件的作用是输出 JNI 层中的相关日志，本质上是调用了 Android 平台的 `__android_log_print` 函数，通过宏定义来输出不同级别的日志，代码如下：

```
#ifndef _LOG_H
#define _LOG_H
```




```
#define LOGN (void) 0

#ifndef WIN32
#include <android/log.h>

#define LOG_VERBOSE    1
#define LOG_DEBUG      2
#define LOG_INFO       3
#define LOG_WARNING    4
#define LOG_ERROR      5
#define LOG_FATAL      6
#define LOG_SILENT     7

#ifndef LOG_TAG
#define LOG_TAG __FILE__
#endif

#ifndef LOG_LEVEL
#define LOG_LEVEL LOG_VERBOSE
#endif

#define LOGP(level, fmt, ...) \
    __android_log_print(level, LOG_TAG, "%s:" fmt, \
        __PRETTY_FUNCTION__, ##__VA_ARGS__)

#if LOG_VERBOSE >= LOG_LEVEL
#define LOGV(fmt, ...) \
    LOGP(ANDROID_LOG_VERBOSE, fmt, ##__VA_ARGS__)
#else
#define LOGV(...) LOGN
#endif

#if LOG_DEBUG >= LOG_LEVEL
#define LOGD(fmt, ...) \
    LOGP(ANDROID_LOG_DEBUG, fmt, ##__VA_ARGS__)
#else
#define LOGD(...) LOGN
#endif

#if LOG_INFO >= LOG_LEVEL
#define LOGI(fmt, ...) \
    LOGP(ANDROID_LOG_INFO, fmt, ##__VA_ARGS__)
#else
#define LOGI(...) LOGN
#endif
```



```
#endif

#if LOG_WARNING >= LOG_LEVEL
#define LOGW(fmt, ...) \
    LOGP(ANDROID_LOG_WARN, fmt, ##__VA_ARGS__)
#else
#define LOGW(...) LOGN
#endif

#if LOG_ERROR >= LOG_LEVEL
#define LOGE(fmt, ...) \
    LOGP(ANDROID_LOG_ERROR, fmt, ##__VA_ARGS__)
#else
#define LOGE(...) LOGN
#endif

#if LOG_FATAL >= LOG_LEVEL
#define LOGF(fmt, ...) \
    LOGP(ANDROID_LOG_FATAL, fmt, ##__VA_ARGS__)
#else
#define LOGF(...) LOGN
#endif

#if LOG_FATAL >= LOG_LEVEL
#define LOGA(condition, fmt, ...) \
    if (!(condition)) \
    { \
        __android_log_assert(condition, LOG_TAG, "(%s:%u) %s: error:%s " fmt, \
        __FILE__, __LINE__, __PRETTY_FUNCTION__, condition, ##__VA_ARGS__); \
    }
#else
#define LOGA(...) LOGN
#endif

#else
#include <stdio.h>
#define LOGP(fmt, ...) printf("%s line:%d " fmt, __FILE__, __LINE__,
##__VA_ARGS__)
#define LOGV(fmt, ...) LOGP(fmt, ##__VA_ARGS__)
#define LOGD(fmt, ...) LOGP(fmt, ##__VA_ARGS__)
#define LOGI(fmt, ...) LOGP(fmt, ##__VA_ARGS__)
#define LOGW(fmt, ...) LOGP(fmt, ##__VA_ARGS__)
#define LOGE(fmt, ...) LOGP(fmt, ##__VA_ARGS__)
#define LOGF(fmt, ...) LOGP(fmt, ##__VA_ARGS__)
```




```
#define LOGA(...) LOGN
#endif // ANDROID_PROJECT
#endif // _LOG_H
```

NativePlayer.c 类的功能有如下几个。

- (1) 获取采集的音视频数据。
- (2) 设置音视频属性。
- (3) 视频进行 H.264 编码, 音频进行 AAC 编码。
- (4) 加入 RTMPPacket 队列, 等待发送线程发送到流媒体服务器上。
- (5) 从队列中不断拉取 RTMPPacket 并发送给流媒体服务器。

需要说明的是, 使用 x264 进行 H.264 编码, 使用 faac 进行 AAC 编码, 需要根据 x264 开源库和 faac 开源库按步骤操作:

```
#include "com_hejunlin_ffmpegpushcamerastream_NativePlayer.h"
#include "log.h"
#include <android/native_window_jni.h>
#include <android/native_window.h>

#include <pthread.h>
#include "doubleQueue.h"
#include "x264.h"
#include "rtmp.h"
#include "faac.h"

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

#define CONNECT_FAILED 101
#define INIT_FAILED 102

//x264 编码输入图像 YUV420P
x264_picture_t pic_in;
x264_picture_t pic_out;
//YUV 个数
int y_len, u_len, v_len;
//x264 编码处理器
x264_t *video_encode_handle;
```




```
unsigned int start_time;
//线程处理
pthread_mutex_t mutex;
pthread_cond_t cond;
//RTMP 流媒体地址
char *rtmp_path;
//是否直播
int is_pushing = FALSE;
//faac 音频编码处理器
faacEncHandle audio_encode_handle;

unsigned long nInputSamples; //输入的采样个数
unsigned long nMaxOutputBytes; //编码输出之后的字节数

//jobject jobj_push_native; //Global ref
//jclass jcls_push_native;
//jmethodID jmid_throw_native_error;
//JavaVM *javaVM;
/**
 * 添加 AAC 头信息
 */
void add_aac_sequence_header(){
    //获取 AAC 头信息的长度
    unsigned char *buf;
    unsigned long len; //长度
    faacEncGetDecoderSpecificInfo(audio_encode_handle, &buf, &len);
    int body_size = 2 + len;
    RTMPPacket *packet = malloc(sizeof(RTMPPacket));
    //RTMPPacket 初始化
    RTMPPacket_Alloc(packet, body_size);
    RTMPPacket_Reset(packet);
    unsigned char * body = packet->m_body;
    //头信息配置
    /*AF 00 + AAC RAW data*/
    body[0] = 0xAF;
    //10 5 SoundFormat(4bits):10 = AAC, SoundRate(2bits):3 = 44kHz,
    //SoundSize(1bit):1 = 16-bit samples, SoundType(1bit):1=Stereo sound
    body[1] = 0x00; //AACPacketType:0 表示 AAC sequence header
    memcpy(&body[2], buf, len); /*spec_buf 是 AAC sequence header 数据*/
    packet->m_packetType = RTMP_PACKET_TYPE_AUDIO;
    packet->m_nBodySize = body_size;
    packet->m_nChannel = 0x04;
    packet->m_hasAbsTimestamp = 0;
```



```
    packet->m_nTimeStamp = 0;
    packet->m_headerType = RTMP_PACKET_SIZE_MEDIUM;
    add_rtmp_packet(packet);
    free(buf);

}

/**
 * 添加 AAC RTMP Packet
 */
void add_aac_body(unsigned char *buf, int len){
    int body_size = 2 + len;
    RTMPPacket *packet = malloc(sizeof(RTMPPacket));
    //RTMPPacket 初始化
    RTMPPacket_Alloc(packet, body_size);
    RTMPPacket_Reset(packet);
    unsigned char * body = packet->m_body;
    //头信息配置
    /*AF 00 + AAC RAW data*/
    body[0] = 0xAF;
    //10 5 SoundFormat(4bits):10=AAC, SoundRate (2bits):3 = 44kHz,
    //SoundSize(1bit):1 = 16-bit samples, SoundType(1bit):1=Stereo sound
    body[1] = 0x01; //AACPacketType:1 表示 AAC raw
    memcpy(&body[2], buf, len); /*spec_buf 是 AAC raw 数据*/
    packet->m_packetType = RTMP_PACKET_TYPE_AUDIO;
    packet->m_nBodySize = body_size;
    packet->m_nChannel = 0x04;
    packet->m_hasAbsTimeStamp = 0;
    packet->m_headerType = RTMP_PACKET_SIZE_LARGE;
    packet->m_nTimeStamp = RTMP_GetTime() - start_time;
    add_rtmp_packet(packet);
}

//获取 JavaVM
/*jint JNI_OnLoad(JavaVM* vm, void* reserved){
    javaVM = vm;
    return JNI_VERSION_1_4;
}*/

/**
 * 向 Java 层发送错误信息
 */
/*void throwNativeError(JNIEnv *env, int code){
```




```
(*env)->CallVoidMethod(env, jobj_push_native, jmid_throw_native_error,
code);
}*/
/**
 * 从队列中不断拉取 RTMPPacket 并发送给流媒体服务器
 */
void *push_thread(void * arg){
//  JNIEnv* env;//获取当前线程 JNIEnv
// (*javaVM)->AttachCurrentThread(javaVM, &env, NULL);
// 建立 RTMP 连接
RTMP *rtmp = RTMP_Alloc();
if(!rtmp){
    LOGE("RTMP 初始化失败");
    goto end;
}
RTMP_Init(rtmp);
rtmp->Link.timeout = 5; //连接超时的时间
//设置流媒体地址
RTMP_SetupURL(rtmp, rtmp_path);
//发布 RTMP 数据流
RTMP_EnableWrite(rtmp);
//建立连接
if(!RTMP_Connect(rtmp, NULL)){
    LOGE("%s", "RTMP 连接失败");
    goto end;
}
//计时
start_time = RTMP_GetTime();
if(!RTMP_ConnectStream(rtmp, 0)){ //连接流
    LOGE("%s", "RTMP ConnectStream failed");
    goto end;
}
is_pushing = TRUE;
//发送 AAC 头信息
//add_aac_sequence_header();
while(is_pushing){
    //发送
    pthread_mutex_lock(&mutex);
    pthread_cond_wait(&cond, &mutex);
    //取出队列中的 RTMPPacket
    RTMPPacket *packet = queue_get_first();
    if(packet){
        queue_delete_first(); //移除
```




```
packet->m_nInfoField2 = rtmp->m_stream_id;
//RTMP 协议, stream_id 数据
int i = RTMP_SendPacket(rtmp, packet, TRUE);
//将 TRUE 放入 librtmp 队列中, 并不立即发送
if(!i){
    LOGE("RTMP 断开");
    RTMPPacket_Free(packet);
    pthread_mutex_unlock(&mutex);
    goto end;
}else{
    LOGI("%s", "rtmp send packet");
}
RTMPPacket_Free(packet);
}

pthread_mutex_unlock(&mutex);
}
end:
LOGI("%s", "释放资源");
free(rtmp_path);
RTMP_Close(rtmp);
RTMP_Free(rtmp);
//(*javaVM)->DetachCurrentThread(javaVM);
return 0;
}

JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
startPush
(JNIEnv *env, jobject jobj, jstring url_jstr){
//jobj(PushNative 对象)
//jobj_push_native = (*env)->NewGlobalRef(env, jobj);

/* jclass jcls_push_native_tmp = (*env)->GetObjectClass(env, jobj);
jcls_push_native = (*env)->NewGlobalRef(env, jcls_push_native_tmp);
if(jcls_push_native_tmp == NULL){
    LOGI("%s", "NULL");
}else{
    LOGI("%s", "not NULL");
}
//PushNative.throwNativeError
jmid_throw_native_error = (*env)->GetMethodID(env, jcls_push_native_tmp,
"throwNativeError", "(I)V");
//初始化的操作 */
```



```
const char* url_cstr = (*env)->GetStringUTFChars(env,url_jstr,NULL);
//复制 url_cstr 内容到 rtmp_path
rtmp_path = malloc(strlen(url_cstr) + 1);
memset(rtmp_path,0,strlen(url_cstr) + 1);
memcpy(rtmp_path,url_cstr,strlen(url_cstr));

//初始化互斥锁与条件变量
pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond,NULL);

//创建队列
create_queue();
//启动消费者线程（从队列中不断拉取 RTMPPacket 并发送给流媒体服务器）
pthread_t push_thread_id;
pthread_create(&push_thread_id, NULL,push_thread, NULL);

(*env)->ReleaseStringUTFChars(env,url_jstr,url_cstr);
}

JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
stopPush(JNIEnv *env, jobject jobj){
    is_pushing = FALSE;
}

JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
release(JNIEnv *env, jobject jobj){
    /*      (*env)->DeleteGlobalRef(env,jcls_push_native);
    (*env)->DeleteGlobalRef(env,jobj_push_native);
    (*env)->DeleteGlobalRef(env,jmid_throw_native_error);*/
}

/**
 * 设置视频参数
 */
JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
setVideoOptions(JNIEnv *env, jobject jobj, jint width, jint height, jint
bitrate, jint fps){
    x264_param_t param;
    //x264_param_default_preset 设置
    x264_param_default_preset(&param,"ultrafast","zerolatency");
    //编码输入的像素格式 YUV420P
    param.i_csp = X264_CSP_I420;
    param.i_width = width;
```




```
param.i_height = height;

y_len = width * height;
u_len = y_len / 4;
v_len = u_len;

//参数 i_rc_method 表示码率控制, 包括 CQP (恒定质量)、CRF (恒定码率)、
//ABR (平均码率) 恒定码率, 会尽量控制在固定码率
param.rc.i_rc_method = X264_RC_CRF;
param.rc.i_bitrate = bitrate / 1000; /* 码率 (比特率, 单位 kb/s)
param.rc.i_vbv_max_bitrate = bitrate / 1000 * 1.2; //瞬时最大码率

//码率控制不是通过 timebase 和 timestamp, 而是通过 fps
param.b_vfr_input = 0;
param.i_fps_num = fps; /*帧率分子
param.i_fps_den = 1; /*帧率分母
param.i_timebase_den = param.i_fps_num;
param.i_timebase_num = param.i_fps_den;
param.i_threads = 1; //并行编码线程数量, 0 默认表示多线程

//是否把 SPS 和 PPS 放入每一个关键帧
//SPS Sequence Parameter Set 是序列参数集, PPS Picture Parameter Set 是图像参数集
//为了提高图像的纠错能力
param.b_repeat_headers = 1;
//设置 Level 级别
param.i_level_idc = 51;
//设置 Profile 档次
//baseline 级别, 没有 B 帧
x264_param_apply_profile(&param, "baseline");

//x264_picture_t (输入图像) 初始化
x264_picture_alloc(&pic_in, param.i_csp, param.i_width, param.i_height);
pic_in.i_pts = 0;
//打开编码器
video_encode_handle = x264_encoder_open(&param);
if(video_encode_handle){
    LOGI("成功打开编码器.....");
}
}

JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
setAudioOptions(JNIEnv *env, jobject jobj, jint sampleRateInHz, jint
numChannels){
```



```

    audio_encode_handle = faacEncOpen(sampleRateInHz, numChannels,
&nInputSamples, &nMaxOutputBytes);
    if(!audio_encode_handle){
        LOGE("音频编码器打开失败");
        return;
    }
    //设置音频编码参数
    faacEncConfigurationPtr p_config = faacEncGetCurrentConfiguration
(audio_encode_handle);
    p_config->mpegVersion = MPEG4;
    p_config->allowMidside = 1;
    p_config->aacObjectType = LOW;
    p_config->outputFormat = 0; //输出是否包含 ADTS 头
    p_config->useTns = 1; //时域噪声控制, 大概就是消爆音
    p_config->useLfe = 0;
    //p_config->inputFormat = FAAC_INPUT_16BIT;
    p_config->quantqual = 100;
    p_config->bandWidth = 0; //频宽
    p_config->shortctl = SHORTCTL_NORMAL;

    if(!faacEncSetConfiguration(audio_encode_handle,p_config)){
        LOGE("%s", "音频编码器配置失败");
        return;
    }

    LOGI("%s", "音频编码器配置成功");
}

/**
 * 加入 RTMPPacket 队列, 等待发送线程发送
 */
void add_rtmp_packet(RTMPPacket *packet){
    pthread_mutex_lock(&mutex);
    if(is_pushing){
        queue_append_last(packet);
    }
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}

/**
 * 发送 H.264 SPS 与 PPS 参数集
 */

```

Android 音视频开发

```

void add_264_sequence_header(unsigned char* pps,unsigned char* sps,int
pps_len,int sps_len){
    int body_size = 16 + sps_len + pps_len;
    //按照 H.264 标准配置 SPS 和 PPS, 共使用 16 字节
    RTMPPacket *packet = malloc(sizeof(RTMPPacket));
    //初始化 RTMPPacket
    RTMPPacket_Alloc(packet,body_size);
    RTMPPacket_Reset(packet);

    unsigned char * body = packet->m_body;
    int i = 0;
    //二进制表示: 00010111
    body[i++] = 0x17;
    //VideoHeaderTag:FrameType(1=key frame)+CodecID(7=AVC)
    body[i++] = 0x00;
    //AVCPacketType = 0 表示设置 AVCDecoderConfigurationRecord
    body[i++] = 0x00;
    body[i++] = 0x00;
    body[i++] = 0x00;

    /*AVCDecoderConfigurationRecord*/
    body[i++] = 0x01;//configurationVersion, 版本为 1
    body[i++] = sps[1];//AVCProfileIndication
    body[i++] = sps[2];//profile_compatibility
    body[i++] = sps[3];//AVCLevelIndication
    body[i++] = 0xFF;//lengthSizeMinusOne, H264 视频中 NALU 的长度, 计算方法是
    //1 + (lengthSizeMinusOne & 3), 实际测试时发现值总为 FF, 计算结果为 4

    /*SPS*/
    body[i++] = 0xE1;//numOfSequenceParameterSets 表示 SPS 的个数, 计算方法是
    //numOfSequenceParameterSets & 0x1F, 实际测试时发现值总为 E1, 计算结果为 1
    body[i++] = (sps_len >> 8) & 0xff;
    //sequenceParameterSetLength 表示 SPS 的长度
    body[i++] = sps_len & 0xff;//sequenceParameterSetNALUnits
    memcpy(&body[i], sps, sps_len);
    i += sps_len;

    /*PPS*/
    body[i++] = 0x01;//numOfPictureParameterSets 表示 PPS 的个数, 计算方法是
    //numOfPictureParameterSets & 0x1F, 实际测试时发现值总为 E1, 计算结果为 1
    body[i++] = (pps_len >> 8) & 0xff;
    //pictureParameterSetLength 表示 PPS 的长度
    body[i++] = (pps_len) & 0xff;//PPS

```



```

memcpy(&body[i], pps, pps_len);
i += pps_len;

//Message Type (消息类型), RTMP_PACKET_TYPE_VIDEO: 0x09
packet->m_packetType = RTMP_PACKET_TYPE_VIDEO;
//Payload Length (Payload 长度)
packet->m_nBodySize = body_size;
//Time Stamp (时间戳): 4 字节
//记录了每一个 tag 相对于第一个 tag (File Header) 的相对时间
//以 ms 为单位, 而 File Header 的 timestamp 永远为 0
packet->m_nTimeStamp = 0;
packet->m_hasAbsTimestamp = 0;
packet->m_nChannel = 0x04; //Channel ID, Audio 和 Video 通道
packet->m_headerType = RTMP_PACKET_SIZE_MEDIUM;
//加入 RTMPPacket
add_rtmp_packet(packet);

}

/**
 * 发送 H.264 帧信息
 */
void add_264_body(unsigned char *buf, int len) {
    //去掉起始码 (界定符)
    if(buf[2] == 0x00){ //00 00 01
        buf += 4;
        len -= 4;
    }else if(buf[2] == 0x01){ //00 00 01
        buf += 3;
        len -= 3;
    }
    int body_size = len + 9;
    RTMPPacket *packet = malloc(sizeof(RTMPPacket));
    RTMPPacket_Alloc(packet, body_size);

    unsigned char * body = packet->m_body;
    //在 NAL 头信息中, type (5 位) 等于 5, 说明这是关键帧 NAL 单元
    //buf[0] NAL Header 与运算, 获取 type, 根据 type 判断关键帧和普通帧
    //00000101 & 00011111 (0x1f) = 00000101
    int type = buf[0] & 0x1f;
    //Inter Frame, 帧间压缩
    body[0] = 0x27;
    //VideoHeaderTag:FrameType(2=Inter Frame)+CodecID(7=AVC)

```


Android 音视频开发

```

//IDR, I 帧图像
if (type == NAL_SLICE_IDR) {
    body[0] = 0x17;
    //VideoHeaderTag:FrameType(1=key frame)+CodecID(7=AVC)
}
//AVCPacketType = 1
body[1] = 0x01; /*nal unit,NALUs (AVCPacketType == 1)*/
body[2] = 0x00; //composition time 0x000000 24bit
body[3] = 0x00;
body[4] = 0x00;

//写入 NALU 信息, 右移 8 位, 一个字节的读取
body[5] = (len >> 24) & 0xff;
body[6] = (len >> 16) & 0xff;
body[7] = (len >> 8) & 0xff;
body[8] = (len) & 0xff;

/*复制数据*/
memcpy(&body[9], buf, len);

packet->m_hasAbsTimestamp = 0;
packet->m_nBodySize = body_size;
packet->m_packetType = RTMP_PACKET_TYPE_VIDEO;
//当前 Packet 的类型为 Video
packet->m_nChannel = 0x04;
packet->m_headerType = RTMP_PACKET_SIZE_LARGE;
//packet->m_nTimeStamp = -1;
packet->m_nTimeStamp = RTMP_GetTime() - start_time;
//记录了每一个 tag 相对于第一个 tag (File Header) 的相对时间
add_rtmp_packet(packet);
}

/**
 * 对采集到的视频数据进行编码
 */
JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
sendVideoPacket(JNIEnv *env, jobject jobj, jbyteArray buffer){
    //将视频数据转为 YUV420P
    //NV21->YUV420P
    jbyte* nv21_buffer = (*env)->GetByteArrayElements(env,buffer,NULL);
    jbyte* u = pic_in.img.plane[1];
    jbyte* v = pic_in.img.plane[2];
    //nv21 4:2:0 Formats, 12 Bits per Pixel

```

```

//nv21 与 yuv420p, y 个数一致, uv 位置对调
//nv21 转 yuv420p  y = w*h,u/v=w*h/4
//nv21 = yvu yuv420p=yuv y=y u=y+1+1 v=y+1
memcpy(pic_in.img.plane[0], nv21_buffer, y_len);
int i;
for (i = 0; i < u_len; i++) {
    *(u + i) = *(nv21_buffer + y_len + i * 2 + 1);
    *(v + i) = *(nv21_buffer + y_len + i * 2);
}

//通过 H.264 编码得到 NALU 数组
x264_nal_t *nal = NULL; //NAL
int n_nal = -1; //NALU 的个数
//进行 H.264 编码
if(x264_encoder_encode(video_encode_handle,&nal, &n_nal, &pic_in,
&pic_out) < 0){
    LOGE("%s","编码失败");
    return;
}
//使用 RTMP 协议将 H.264 编码的视频数据发送给流媒体服务器
//帧分为关键帧和普通帧, 为了提高画面的纠错率, 关键帧应包含 SPS 和 PPS 数据
int sps_len , pps_len;
unsigned char sps[100];
unsigned char pps[100];
memset(sps,0,100);
memset(pps,0,100);
pic_in.i_pts += 1; //顺序累加
//遍历 NALU 数组, 根据 NALU 的类型判断
for(i=0; i < n_nal; i++){
    if(nal[i].i_type == NAL_SPS){
        //复制 SPS 数据
        sps_len = nal[i].i_payload - 4;
        memcpy(sps,nal[i].p_payload + 4,sps_len); //不复制 4 字节起始码
    }else if(nal[i].i_type == NAL_PPS){
        //复制 PPS 数据
        pps_len = nal[i].i_payload - 4;
        memcpy(pps,nal[i].p_payload + 4,pps_len); //不复制 4 字节起始码
        //发送序列信息
        //H.264 关键帧会包含 SPS 和 PPS 数据
        add_264_sequence_header(pps,sps,pps_len,sps_len);
    }else{
        //发送帧信息
        add_264_body(nal[i].p_payload,nal[i].i_payload);
    }
}

```


Android 音视频开发

```

    }
}

JNIEXPORT void JNICALL Java_com_hejunlin_ffmpegpushcamerastream_NativePlayer_
sendAudioPacket(JNIEnv *env, jobject jobj, jbyteArray buffer, jint len){
    int *pcmbuf;
    unsigned char *bitbuf;
    jbyte* b_buffer = (*env)->GetByteArrayElements(env, buffer, 0);
    pcmbuf = (short*) malloc(nInputSamples * sizeof(int));
    bitbuf = (unsigned char*) malloc(nMaxOutputBytes * sizeof(unsigned char));
    int nByteCount = 0;
    unsigned int nBufferSize = (unsigned int) len / 2;
    unsigned short* buf = (unsigned short*) b_buffer;
    while (nByteCount < nBufferSize) {
        int audioLength = nInputSamples;
        if ((nByteCount + nInputSamples) >= nBufferSize) {
            audioLength = nBufferSize - nByteCount;
        }
        int i;
        for (i = 0; i < audioLength; i++) {
            //每次从实时的 PCM 音频队列中读出量化位数为 8 的 PCM 数据
            int s = ((int16_t *) buf + nByteCount)[i];
            pcmbuf[i] = s << 8; //用 8 个二进制位来表示一个采样量化点（模数转换）
        }
        nByteCount += nInputSamples;
        //利用 FAAC 进行编码，pcmbuf 为转换后的 PCM 流数据，audioLength 为调用
        //faacEncOpen 时得到的输入采样数，bitbuf 为编码后的数据 buff，nMaxOutputBytes 为
        //调用 faacEncOpen 时得到的最大输出字节数
        int byteslen = faacEncEncode(audio_encode_handle, pcmbuf, audioLength,
        bitbuf, nMaxOutputBytes);
        if (byteslen < 1) {
            continue;
        }
        add_aac_body(bitbuf, byteslen);
        //从 bitbuf 中得到编码后的 AAC 数据流，放到数据队列中
    }
    (*env)->ReleaseByteArrayElements(env, buffer, b_buffer, NULL);
    if (bitbuf)
        free(bitbuf);
    if (pcmbuf)
        free(pcmbuf);
}

```

手机直播推流如图 9-15 所示。



图 9-15 手机直播推流

这里用 VLC Mac OS 客户端及 Android 平台的ijkplayer进行其他客户端拉流播放，效果如图 9-16 所示。

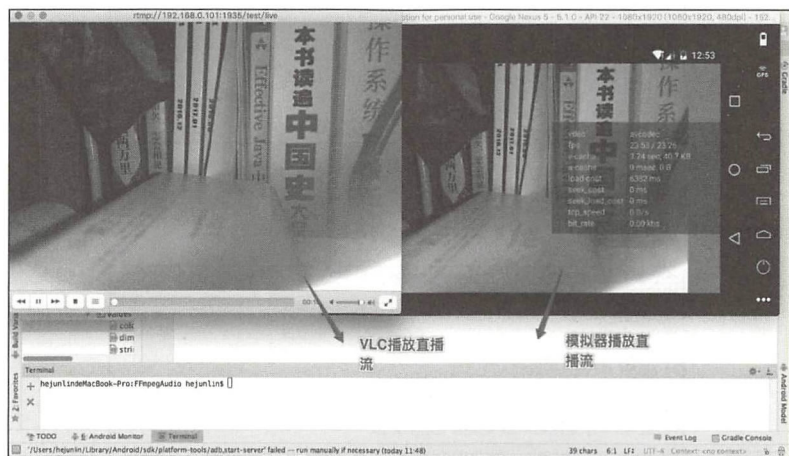


图 9-16 客户端拉流播放

9.4 流媒体服务器搭建

Nginx 是一款非常优秀的开源服务器，用它来做 HLS 或者 RTMP 流媒体服务器是非常不错的选择。Nginx 的安装步骤如下。

Android 音视频开发

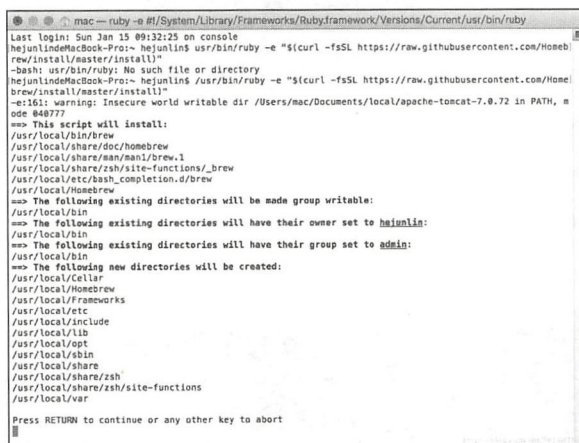
1. 安装 Homebrew

Homebrew 简称 brew，是 Mac OS X 上的软件包管理工具，能在 Mac 中方便地安装软件或者卸载软件。可以说 Homebrew 就是 Mac 下的 apt-get、yum 神器，因为 Mac 本身也是基于 UNIX 内核的。

如果你的机器上没有安装 Homebrew，可以通过如下命令进行安装：

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

安装 Homebrew，如图 9-17 所示。



```
mac - ruby -e #!/System/Library/Frameworks/Ruby.framework/Versions/Current/usr/bin/ruby
Last login: Sun Jan 15 09:32:25 on console
hejunlin@hejunlin-MacBook-Pro:~$ hejunlin$ /usr/bin/ruby -e "$curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install"
-bash: /usr/bin/ruby: No such file or directory
hejunlin@hejunlin-MacBook-Pro:~$ hejunlin$ /usr/bin/ruby -e "$curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install"
-e:161: warning: Insecure world writable dir /Users/mac/Documents/local/apache-tomcat-7.0.72 in PATH, mode 040777
==> This script will install:
/usr/local/bin/brew
/usr/local/share/doc/homebrew
/usr/local/share/man/man1/brew.1
/usr/local/share/zsh/site-functions/_brew
/usr/local/etc/bash_completion.d/brew
/usr/local/Homebrew
==> The following existing directories will be made group writable:
/usr/local/bin
==> The following existing directories will have their owner set to hejunlin:
/usr/local/bin
==> The following existing directories will have their group set to admin:
/usr/local/bin
==> The following new directories will be created:
/usr/local/Cellar
/usr/local/Homebrew
/usr/local/Frameworks
/usr/local/etc
/usr/local/include
/usr/local/lib
/usr/local/opt
/usr/local/sbin
/usr/local/share
/usr/local/share/zsh
/usr/local/share/zsh/site-functions
/usr/local/var
Press RETURN to continue or any other key to abort
```

图 9-17 安装 Homebrew

安装 Homebrew 的过程，如图 9-18 所示。



```
mac - git - ruby -e #!/System/Library/Frameworks/Ruby.framework/Versions/Current/usr/bin/ruby
/usr/local/include
/usr/local/lib
/usr/local/opt
/usr/local/sbin
/usr/local/share
/usr/local/share/zsh
/usr/local/share/zsh/site-functions
/usr/local/var
Press RETURN to continue or any other key to abort
==> /usr/bin/sudo /bin/chmod g-rwx /usr/local/bin
Password:
==> /usr/bin/sudo /bin/chmod g-rwx /usr/local/bin
==> /usr/bin/sudo /usr/bin/chown hejunlin /usr/local/bin
==> /usr/bin/sudo /usr/bin/chgrp admin /usr/local/bin
==> /usr/bin/sudo /bin/mkdir -p /usr/local/Cellar /usr/local/Homebrew /usr/local/Frameworks /usr/local/etc
/usr/local/include /usr/local/lib /usr/local/opt /usr/local/sbin /usr/local/share /usr/local/share/zsh
/usr/local/share/zsh/site-functions /usr/local/var
==> /usr/bin/sudo /bin/chmod g-rwx /usr/local/Cellar /usr/local/Homebrew /usr/local/Frameworks /usr/local/etc
/usr/local/include /usr/local/lib /usr/local/opt /usr/local/sbin /usr/local/share /usr/local/share/zsh
/usr/local/share/zsh/site-functions /usr/local/var
==> /usr/bin/sudo /bin/chmod 755 /usr/local/share/zsh /usr/local/share/zsh/site-functions
==> /usr/bin/sudo /usr/bin/chown hejunlin /usr/local/Cellar /usr/local/Homebrew /usr/local/Frameworks /usr/local/etc
/usr/local/include /usr/local/lib /usr/local/opt /usr/local/sbin /usr/local/share /usr/local/share/zsh
/usr/local/share/zsh/site-functions /usr/local/var
==> /usr/bin/sudo /usr/bin/chgrp admin /usr/local/Cellar /usr/local/Homebrew /usr/local/Frameworks /usr/local/etc
/usr/local/include /usr/local/lib /usr/local/opt /usr/local/sbin /usr/local/share /usr/local/share/zsh
/usr/local/share/zsh/site-functions /usr/local/var
==> /usr/bin/sudo /bin/mkdir -p /Users/mac/Library/Caches/Homebrew
==> /usr/bin/sudo /usr/bin/chown hejunlin /Users/mac/Library/Caches/Homebrew
==> /usr/bin/sudo /bin/mkdir -p /Library/Caches/Homebrew
==> /usr/bin/sudo /bin/chmod g-rwx /Library/Caches/Homebrew
==> /usr/bin/sudo /usr/bin/chown hejunlin /Library/Caches/Homebrew
==> Downloading and installing Homebrew...
remote: Counting objects: 4471, done.
remote: Compressing objects: 100% (2963/2963), done.
Receiving objects: 60% (2683/4471), 908.00 KiB | 228.00 KiB/s
```

图 9-18 安装 Homebrew 的过程

安装好 Homebrew，如图 9-19 所示。

```

mac -- bash -- 103x38
* [new tag] 1.0.5 -> 1.0.5
* [new tag] 1.0.6 -> 1.0.6
* [new tag] 1.0.7 -> 1.0.7
* [new tag] 1.0.8 -> 1.0.8
* [new tag] 1.0.9 -> 1.0.9
* [new tag] 1.1.0 -> 1.1.0
* [new tag] 1.1.1 -> 1.1.1
* [new tag] 1.1.2 -> 1.1.2
* [new tag] 1.1.3 -> 1.1.3
* [new tag] 1.1.4 -> 1.1.4
* [new tag] 1.1.5 -> 1.1.5
* [new tag] 1.1.6 -> 1.1.6
* [new tag] 1.1.7 -> 1.1.7
HEAD is now at 1296874 Merge pull request #1835 from ilovezfs/unstable-whitelist
==> Tapping homebrew/core
Cloning into '/usr/local/Homebrew/Library/Taps/homebrew/homebrew-core'...
remote: Counting objects: 3829, done.
remote: Compressing objects: 100% (3789/3789), done.
remote: Total 3829 (delta 24), reused 337 (delta 7), pack-reused 8
Receiving objects: 100% (3829/3829), 3.06 MiB | 183.00 KiB/s, done.
Resolving deltas: 100% (24/24), done.
Checking connectivity... done.
Tapped 3784 formulae (3,857 files, 9.6MiB)
==> Cleaning up /Library/Caches/Homebrew...
==> Migrating /Library/Caches/Homebrew to /Users/mac/Library/Caches/Homebrew...
==> Deleting /Library/Caches/Homebrew...
Already up-to-date.
==> Installation successful!

==> Homebrew has enabled anonymous aggregate user behaviour analytics.
Read the analytics documentation (and how to opt-out) here:
https://git.io/brew-analytics

==> Next steps:
- Run 'brew help' to get started
- Further documentation:
https://git.io/brew-docs
hejunlindeMacBook-Pro:~ hejunlin$

```

图 9-19 安装好 Homebrew

2. 安装 Nginx 服务器

增加对 Nginx 的扩展。也就是从 GitHub 上下载，Homebrew 对 Nginx 扩展对应的命令如下：

```
brew tap homebrew/nginx
```

安装 Nginx，如图 9-20 所示。

```

hejunlindeMacBook-Pro:~ hejunlin$ brew tap homebrew/nginx
Updating Homebrew...
==> Tapping homebrew/nginx
Cloning into '/usr/local/Homebrew/Library/Taps/homebrew/homebrew-nginx'...
remote: Counting objects: 70, done.
remote: Compressing objects: 100% (69/69), done.
remote: Total 70 (delta 1), reused 51 (delta 1), pack-reused 0
Unpacking objects: 100% (70/70), done.
Checking connectivity... done.
Tapped 61 formulae (157 files, 125.9K)
hejunlindeMacBook-Pro:~ hejunlin$

```

图 9-20 安装 Nginx

3. 安装 Nginx 服务器和 rtmp 模块

安装命令如下：

```
brew install nginx-full --with-rtmp-module
```

这个安装过程耗时相对较长。通过以上步骤 Nginx 服务器和 rtmp 模块就安装好了，下面开始配置 Nginx 的 rtmp 模块，如图 9-21 所示。


```

hejunlindeMacBook-Pro:~ hejunlin$ brew install nginx-full --with-rtmp-module
Updating Homebrew...
==> Installing nginx-full from homebrew/nginx
==> Installing dependencies for homebrew/nginx/nginx-full: pcre, openssl, rtmp-nginx-module
==> Installing homebrew/nginx/nginx-full dependency: pcre
==> Downloading https://homebrew.bintray.com/bottles/pcre-8.39.el_capitan.bottle.tar.gz
##### 72.5%

```

图 9-21 配置 rtmp 模块

由于网络原因，中间可能会失败，安装过程如图 9-22 所示。

```

mac -- bash -- 103x38
==> Downloading https://homebrew.bintray.com/bottles/openssl-1.0.2j.el_capitan.bottle.tar.gz
##### 100.0%
==> Pouring openssl-1.0.2j.el_capitan.bottle.tar.gz
==> Using the sandbox
==> Caveats
A CA file has been bootstrapped using certificates from the SystemRoots
keychain. To add additional certificates (e.g. the certificates added in
the System keychain), place .pem files in
  /usr/local/etc/openssl/certs
and run
  /usr/local/opt/openssl/bin/c_rehash
This formula is keg-only, which means it was not symlinked into /usr/local.
Apple has deprecated use of OpenSSL in favor of its own TLS and crypto libraries
Generally there are no consequences of this for you. If you build your
own software and it requires this formula, you'll need to add to your
build variables:
  LDFLAGS: -L/usr/local/opt/openssl/lib
  CPPFLAGS: -I/usr/local/opt/openssl/include
==> Summary
  /usr/local/Cellar/openssl/1.0.2j: 1,695 files, 12M
==> Installing homebrew/nginx/nginx-full dependency: rtmp-nginx-module
==> Downloading https://github.com/sergey-dryabzhinsky/nginx-rtmp-module/archive/v1.1.7.10.tar.gz
==> Downloading from https://codeload.github.com/sergey-dryabzhinsky/nginx-rtmp-module/tar.gz/v1.1.7.10
##### 100.0%
  /usr/local/Cellar/nginx-rtmp-module/1.1.7.10: 92 files, 1.4M, built in 25 seconds
==> Installing homebrew/nginx/nginx-full --with-rtmp-module
==> Downloading https://nginx.org/download/nginx-1.10.2.tar.gz
##### 21.6%
curl: (56) SSLRead() return error -9806
Error: failed to download resource "nginx-full"
Download failed: https://nginx.org/download/nginx-1.10.2.tar.gz
hejunlindeMacBook-Pro:~ hejunlin$

```

图 9-22 安装 nginx-full 失败过程

这时可以重新尝试一下，安装好的界面如图 9-23 所示。

```

mac -- bash -- 80x24
nginx will load all files in /usr/local/etc/nginx/servers/.

- Tips -
Run port 80:
$ sudo chown root:wheel /usr/local/opt/nginx-full/bin/nginx
$ sudo chmod u+s /usr/local/opt/nginx-full/bin/nginx
Reload config:
$ nginx -s reload
Reopen Logfile:
$ nginx -s reopen
Stop process:
$ nginx -s stop
Waiting on exit process
$ nginx -s quit

To have launchd start homebrew/nginx/nginx-full now and restart at login:
brew services start homebrew/nginx/nginx-full
Or, if you don't want/need a background service you can just run:
nginx
==> Summary
  /usr/local/Cellar/nginx-full/1.10.2: 8 files, 1.2M, built in 5 minutes 26 seconds
hejunlindeMacBook-Pro:~ hejunlin$

```

图 9-23 安装好 nginx-full

接下来运行如下命令看看 Nginx 的安装位置：

```
brew info nginx-full
```

执行上面的命令后我们可以看到如图 9-24 所示的信息。



```

--with-xsltproc-module
    Build with XSLT Transformations support
--without-openssl
    Build without openssl support
--devel
    Install development version 1.11.8
--HEAD
    Install HEAD version
==> Caveats
Docroot is: /usr/local/var/www

The default port has been set in /usr/local/etc/nginx/nginx.conf to 8080 so that
nginx can run without sudo.

nginx will load all files in /usr/local/etc/nginx/servers/.

- Tips -
Run port 80:
$ sudo chown root:wheel /usr/local/opt/nginx-full/bin/nginx
$ sudo chmod u+s /usr/local/opt/nginx-full/bin/nginx
Reload config:
$ nginx -s reload
Reopen Logfile:
$ nginx -s reopen

```

图 9-24 查看 Nginx 的安装位置

下面介绍一些 Nginx 常用命令。

- nginx -s reload: 表示重新加载配置文件。
- nginx -s reopen: 再次打开日志文件。
- nginx -s stop: 停止服务器。
- nginx -s quit: 退出服务器。

Nginx 文件目录如下。

- Nginx 的安装位置为 /usr/local/Cellar/nginx-full/1.10.1/bin/nginx。
- Nginx 配置文件所在的位置为 /usr/local/etc/nginx/nginx.conf。
- Nginx 服务器根目录所在的位置为 /usr/local/var/www。

执行下面的命令，测试是否能成功启动 Nginx 服务：

```
nginx
```

在浏览器地址栏中输入 <http://localhost:8080>，这样就可以看到如图 9-25 所示的界面。

4. 配置 rtmp

我们先进入 Nginx 目录，用 ls 命令查看其文件，并用 cat 命令读取 nginx.conf 文件，如图 9-26 所示。

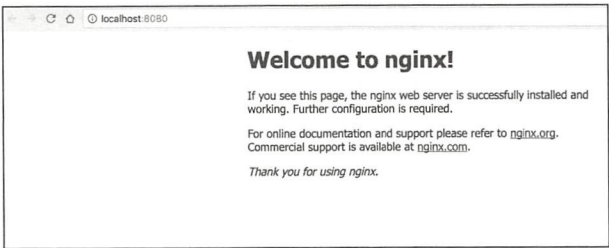


图 9-25 成功启动 Nginx 服务的界面

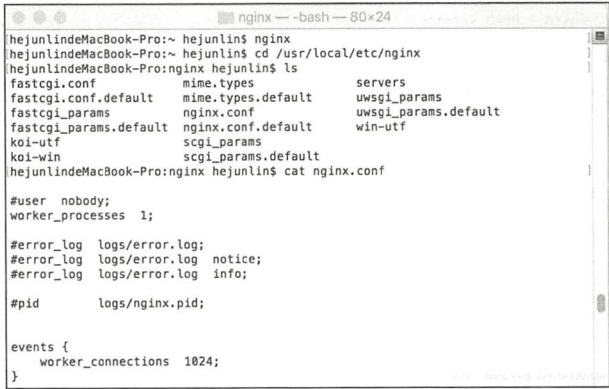


图 9-26 查看 Nginx 配置文件

用 Notepad++打开 nginx.conf，找到usr/local/etc/nginx/nginx.conf文件，就可以打开了。

或者打开 Finder，执行 Mac 的 Shift + command + G 组合键前往，用记事本工具打开 nginx.conf，如图 9-27 所示。

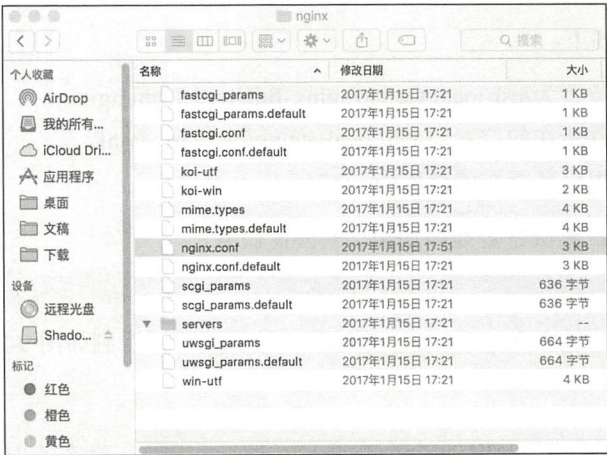


图 9-27 查看 Nginx 配置文件


```
http {
    .....//省略原始配置文件中的内容
}
```

在 http 节点下面（也就是文件的尾部）加上 rtmp 配置：

```
rtmp {
    server {
        listen 1935;
        application test {
            live on;
            record off;
        }
    }
}
```

参数说明如下。

- rtmp 表示协议名称。
- server 说明内部是服务器相关配置。
- listen 表示监听的端口号，RTMP 协议的默认端口号是 1935。
- application 访问的应用路径是 zbcsc。
- live on 表示开启实时流直播。
- record off 表示不记录数据。

最终的配置文件内容如图 9-28 所示。

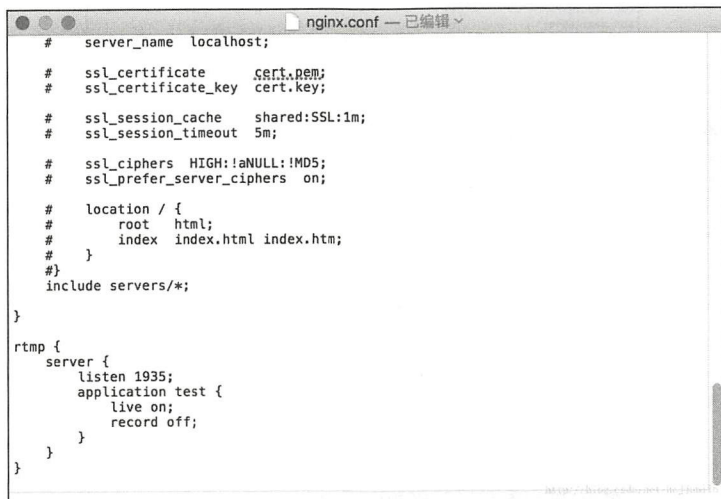


图 9-28 最终的配置文件内容

5. 保存文件后, 重新加载 Nginx 的配置文件 (这一步很重要)

重新加载 Nginx 配置文件的命令如下:

```
nginx -s reload
```

9.5 FFmpeg 推流到流媒体服务器的过程

在 Mac OS 上编译 FFmpeg-3.1.3 源码, 得到可安装文件并进行安装。安装完成后执行命令行, 如 `ffmpeg -version`, 可以看到 FFmpeg 的版本信息。

(1) 复制一份 FFmpeg-3.1.3 源码到一个新建目录下, 并通过命令行进入该目录。

(2) 配置编译选项生成 `makefile`, 在命令行中输入 `./configure --disable-yasm --enable-shared`。生成配置文件, 如图 9-29 所示。



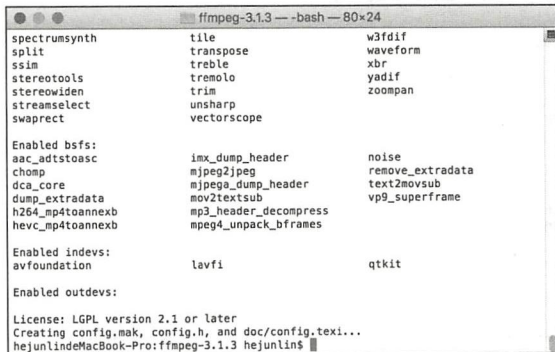
```

hejunlin@MacBook-Pro:ffmpeg-3.1.3 hejunlin$ ./configure --disable-yasm --enable-
-shared
install prefix      /usr/local
source path        .
C compiler          gcc
C library
ARCH                x86_64 (generic)
big-endian          no
runtime cpu detection yes
yasm                no
MMX enabled         yes
MMXEXT enabled      yes
3DNow! enabled      yes
3DNow! extended enabled yes
SSE enabled         yes
SSE3 enabled        yes
AESNI enabled       yes
AVX enabled         yes
XOP enabled         yes
FMA3 enabled        yes
FMA4 enabled        yes
i686 features enabled yes
CMOV is fast        yes
EBX available       yes

```

图 9-29 生成配置文件

最终生成配置文件, 如图 9-30 所示。



```

spectrumynth        tile                w3fdif
split                transpose           waveform
ssim                 treble              xbr
stereotools          tremolo             yadif
stereowiden          trim                zoompan
streamselect         unsharp
swaprect             vectorscope

Enabled bsfs:
aac_adtstoasc        imx_dump_header     noise
chomp                mjpeg2peg            remove_extradata
dca_core              mjpeg9_dump_header  text2movsub
dump_extradata        mov2textsub          vp9_superframe
h264_mp4toannexb     mp3_header_decompress
hevc_mp4toannexb     mpeg4_unpack_bframes

Enabled indevs:
avfoundation          lavfi                qtkit

Enabled outdevs:

License: LGPL version 2.1 or later
Creating config.mak, config.h, and doc/config.texi...
hejunlin@MacBook-Pro:ffmpeg-3.1.3 hejunlin$

```

图 9-30 最终生成配置文件

(3) 执行 make。

make 的编译过程如图 9-31 所示。

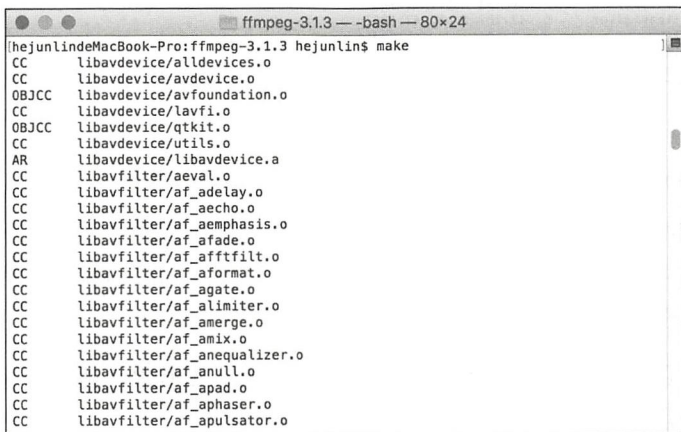


图 9-31 make 的编译过程

(4) 编译完成后，都要执行 `sudo make install` 命令，安装 FFmpeg 工具到默认的 `/usr/local` 目录下的相应位置（在 Mac OS X 下则不推荐使用 `/usr` 目录）。因此不要在配置时指定 `--prefix`，而是使用默认的 `/usr/local` 目录前缀即可，如图 9-32 所示。

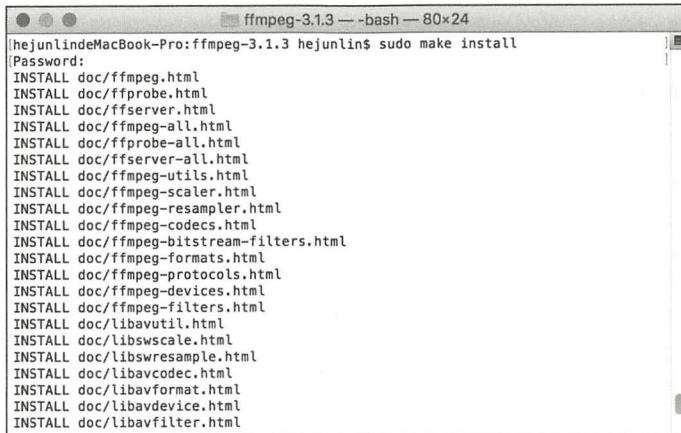


图 9-32 make install 安装

(5) 测试 ffmpeg 命令。

测试 FFmpeg 版本如图 9-33 所示。


```

hejunlindeMacBook-Pro:ffmpeg-3.1.3 hejunlin$ ffmpeg -version
ffmpeg version 3.1.3 Copyright (c) 2000-2016 the FFmpeg developers
built with Apple LLVM version 8.0.0 (clang-800.0.42.1)
configuration: --disable-yasm --enable-shared
libavutil      55. 28.100 / 55. 28.100
libavcodec     57. 48.101 / 57. 48.101
libavformat    57. 41.100 / 57. 41.100
libavdevice    57.  0.101 / 57.  0.101
libavfilter    6. 47.100 /  6. 47.100
libswscale     4.  1.100 /  4.  1.100
libswresample  2.  1.100 /  2.  1.100
hejunlindeMacBook-Pro:ffmpeg-3.1.3 hejunlin$

```

图 9-33 测试 FFmpeg 版本

可以看到 FFmpeg-3.1.3 已经安装好了。

补充一点，还有一种简单的方式，即直接执行 `brew install ffmpeg` 命令，也可以安装 FFmpeg，不过安装的是最新镜像中的最新版本，可以是 FFmpeg-3.3 等。由于要下载版本代码，因此这种方式比较慢。

测试完成后，可以准备一个视频文件作为将要推流的数据源，然后安装一个支持 RTMP 协议的视频播放器。Mac 下可以用 VLC，请读者自行下载。

通过执行 `ffmpeg` 命令开始推流（注意，推流之前要确认 Nginx 是否已启动，可以直接使用 `nginx` 命令）。

推流命令如下：

```

ffmpeg -re -i /Users/mac/Downloads/田埂上的梦.mp4 -vcodec copy -f flv rtmp://localhost:1935/test/live

```

上面命令里的 `test` 是之前的配置文件中配置的 `application` 的名称，后面的 `live` 则应根据自己的需要去写。FFmpeg 推流如图 9-34 所示。

```

ffmpeg-3.1.3 -- ffmpeg -re -i ~/Downloads/田埂上的梦.mp4 -vcodec copy -f flv rt...
hejunlindeMacBook-Pro:ffmpeg-3.1.3 hejunlin$ nginx
hejunlindeMacBook-Pro:ffmpeg-3.1.3 hejunlin$ ffmpeg -re -i /Users/hejunlin/Downl
oads/田埂上的梦.mp4 -vcodec copy -f flv rtmp://localhost:1935/test/live
ffmpeg version 3.1.3 Copyright (c) 2000-2016 the FFmpeg developers
built with Apple LLVM version 8.0.0 (clang-800.0.42.1)
configuration: --disable-yasm --enable-shared
libavutil      55. 28.100 / 55. 28.100
libavcodec     57. 48.101 / 57. 48.101
libavformat    57. 41.100 / 57. 41.100
libavdevice    57.  0.101 / 57.  0.101
libavfilter    6. 47.100 /  6. 47.100
libswscale     4.  1.100 /  4.  1.100
libswresample  2.  1.100 /  2.  1.100
Input #0: mov,mp4,m4a,3gp,3g2,mj2, from '/Users/hejunlin/Downloads/田埂上的梦.m
p4':
Metadata:
  major_brand      : isom
  minor_version   : 1
  compatible_brands: isom
  creation_time   : 2013-10-18 06:21:55
Duration: 00:06:35.19, start: 0.000000, bitrate: 498 kb/s
Stream #0:0(und): Audio: aac (LC) (mp4a / 0x6134706D), 44100 Hz, stereo, flt
p, 63 kb/s (default)
Metadata:
  creation_time   : 2013-10-18 06:21:55
  handler_name    : GPAC ISO Audio Handler
Stream #0:1(und): Video: h264 (Constrained Baseline) (avc1 / 0x31637661), yu
v420p(tv, smpte170m), 640x400 [Sar 1:1 DAR 8:5], 431 kb/s, 23.98 fps, 23.98 tbr,
48k tbn, 47.95 tbc (default)
Metadata:
  handler_name    : Video Media Handler
  encoder        : AVC Coding

```

图 9-34 FFmpeg 推流

最后用 VLC 工具看看效果, 通过 open network 输入对应的推流地址, 播放效果如图 9-35 所示。



图 9-35 VLC 客户端拉取服务器的数据流播放

9.6 直播优化那些事

为了获得更好的体验, 需要对直播的很多环节进行优化, 达到开播快、延时短等实时性要求。

9.6.1 卡顿优化

视频直播卡顿, 需要从以下几个方面分析。

- 设备。
- 视频流。
- 网络。

1. 设备

在设备方面, 如设备的配置太低, 解码将导致卡顿。可优化点: 升级硬件设备、升级软件设备、提高兼容性和容错率等; 降低视频码率, 选择流畅或标清画质进行播放; 增大缓冲区, 缓解因网络或解码不稳定引起的卡顿。

2. 视频流

音视频不同步也会导致卡顿, 如声音连续, 画面静止, 另外则可能是视频流参数配置有问题。音视频不同步有两种情况, 一种是推流时音画就不同步, 另一种是拉流播放时出现了音画

不同步，两种音视频不同步的情况都需要对相应的同步算法进行处理。

可优化点如下。

- 设置合理的帧率（如 25fps）、码率、分辨率、关键帧的间隔。
- 按照视频编码（H.264/H.265）标准方案，编码完整数据。

3. 网络

网络状况不好。当网速较差时，下载数据较慢，用于播放的流数据很少。客户端可以通过监控网络状态变化来降低码流，达到正常播放，减少出现卡顿的情况。

9.6.2 延时优化

低延时，是做播放优化所追求的目标之一。低延时需要多种策略配合使用，才会有好的效果。首先需要分析哪些环节导致延时，然后才对症下药。在推流时，采集到音视频数据后，需要对其进行处理和编码，这个过程是有延时的，其次在封包后通过流媒体协议推流时，受网络状态的影响也会出现延时。在播放端，涉及拉流和解码延时。

1. 处理数据延时优化

在采集到音视频数据后，需要增加一些视频效果，如美颜、水印、滤镜、加贴纸效果等，这些操作需要 GPU 处理，这样可以大大减少处理数据的耗时。

2. 编码延时优化

（1）编码前丢帧，减少编码耗时，比如可以丢 B 帧。如果有 B 帧，解码时需要依赖于前后视频帧，会增加延时。在编码前，丢弃 B 帧，不仅能减少编码耗时，降低带宽开销，还可以减少后续的解码延时。

（2）如果使用了 FFmpeg 库，要降低 probesize 和 analyze duration 两个参数的值，这两个值用于设置视频帧信息监测大小和监测时长，这两个值越大对编码延时的影响越大。

（3）固定码率编码 CBR 可以在一定程度上消除网络抖动的影响，而如果能够使用可变码率编码 VBR，则可以节省一些不必要的网络带宽，减少一定的延时。因此建议尽量使用 VBR 进行编码。

3. 传输协议优化

选择 HTTP-FLV 协议或者 RTMP 协议，这两个协议延时低，可以在网络请求和响应时，快速传输数据，减少延时。

4. 传输网络优化

(1) 在 CDN 节点中缓存当前推流的 GOP，配合播放器端优化视频首开时间。

(2) 后台实时记录每个视频流流向每个环节时的秒级帧率和码率，实时监控码率和帧率的波动。

(3) 通过直连 IP 方式，访问直播地址，这样可以减少域名带来的 DNS 解析耗时。

5. 推流、播放优化

播放端使用动态 Buffer 策略，通常在播放器内部要设置一定的 Buffer，以便能在播放过程中缓冲一定的数据。在播放器开启的时候采用非常小甚至零缓存的策略，通过下载首个视频片段（如 m3u8 文件中的第一个 TS 分片）的耗时来决定下一个时间片的缓存大小，同时在播放过程中实时监测当前网络，实时调整播放过程中缓存的大小。这样既可做到极短的首开时间，而且能够尽量消除网络抖动造成的影响。

9.6.3 数据代理优化

通常对于播放器请求数据，如果是直接请求 CDN，可能会出现一些播放控制不是很理想的情况，如直播中网络状况不好，就会有 loading；如果是在播放器后端之前加上数据代理，数据代理就可以缓冲一部分数据，供播放器源源不断地进行播放，当网络状况不好时，可持续播放的能力越强，loading 出现得就越晚。

图 9-36 是一种本地数据代理方案的整体结构图。

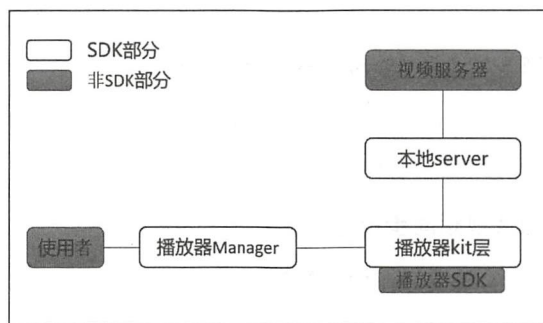


图 9-36 一种本地数据代理方案的整体结构图

播放器 Manager 主要给项目使用，提供一些简单的接口，屏蔽所有能屏蔽的细节。

播放器 kit 层是对播放器 SDK 的简单封装，方便使用。播放器 SDK 是基于 FFmpeg 写的独立的 SDK。

本地 server 是一个 HTTP 代理服务，下面将描述这个模块。

首先，播放器 SDK 并不直接连接视频服务器，而是先连接本地 server，然后由本地 server 连接视频服务器，本地 server 收到服务器端的视频数据，再转发给播放器，相当于一个转发功能。不断接收来自播放器的请求，然后下载数据，再返回给播放器。本地 server 结构图如图 9-37 所示。

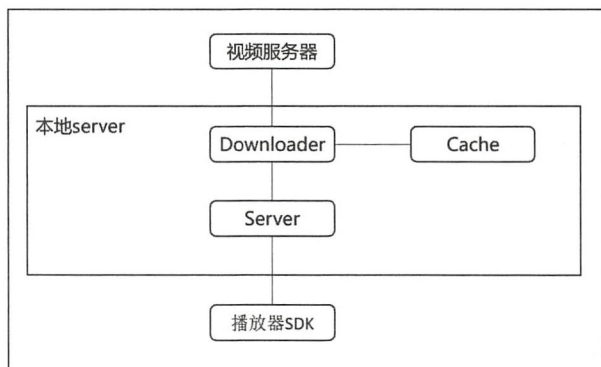


图 9-37 本地 server 结构图

9.6.4 首屏秒开优化

在直播开发中，为了提升开播体验，通常需要做一些秒开优化。几乎直播 App 都能达到秒开。

秒开需要优化的内容如下。

1. 服务器边缘节点缓存 GOP (Group of Pictures)

在播放器拉流过程中，无须下载整个缓存的 GOP，由于 GOP 的第 1 帧通常是关键帧，下载并解码播放首个关键帧，即可达到首帧秒开。

2. 画面首帧渲染不做音视频同步校验

为了达到更快渲染画面的效果，在做首帧渲染时，不做音视频同步，后面再进行同步。

3. 增加上行、下行带宽探测接口，当带宽不满足时降低视频质量，即降低码率

主要是考虑到起播时，网络如果比较差，需要对带宽进行探测，以便使用合适的码率播放视频，缩短首帧渲染时间。

4. 对一些业务逻辑和 UI 进行预加载或者延迟加载

这样做的目的主要是，确保在渲染首帧时，其他网络请求或者 UI 加载不会影响首帧加载时间。

9.6.5 弱网优化

弱网优化，可以从推流端和播放端两个方向进行优化。

首先介绍推流端优化，如下所述。

1. 根据上行带宽的状况来动态调整码率、帧率、分辨率

由于直播的实时性要求，中间过程中可能遭遇网络变差，为了保证直播流延续，需要根据上行带宽来适当降低码率、帧率、分辨率。不同的带宽对应一定范围的码率、帧率值，不至于太低，也不会太高导致不流畅。

2. 推流端使用 H.265 编码推流

如果使用 H.265 编码进行推流，可以节省 40% 宽带，但是并不是所有手机都支持用 H.265 编码格式播放，这时可以通过大数据收集支持 H.265 编码的手机型号，并针对手机型号进行推流。

3. 播放端弱网优化

不同播放内核的弱网优化策略是不同的。如在 VLC 中，可以考虑在网络状况很差的情况下，将视频画面卡住，待 Buffering 数据填充 5s 后再进行播放。对于 FFmpeg，也是根据其在弱网环境下的表现来设置具体的优化策略的。

9.6.6 运营商劫持优化

运营商劫持是常见的网络访问时的问题，通常遇到的有 DNS 劫持和 HTTP 劫持。DNS 劫持主要是让你重定向到其他网站，比如你想正常访问直播后台的某主播房间的地址，正常 DNS 解析 IP 应该是直播后台的 IP，但是中间者修改这个 DNS 解析结果，返回了一个非直播后台的地址，用户就会和这个非直播后台的地址建立数据传输。DNS 劫持的主要预防方法是使用固定的 DNS 地址，比如 8.8.8.8 114.114.114.114。HTTP 劫持主要是运营商为了非法牟利，当发现你的请求是 HTTP 请求时，就往里面插入一些广告和无关紧要的东西，如图 9-38 所示。

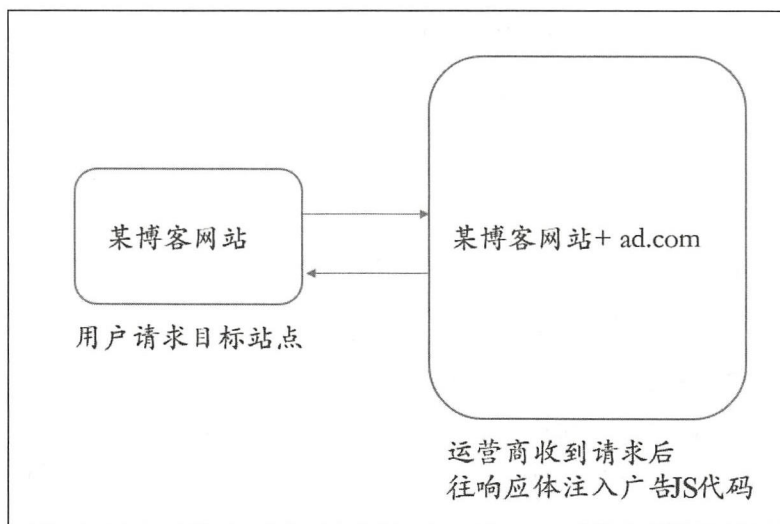


图 9-38 运营商劫持

图 9-38 中的用户本来想上某博客网站，而运营商发现该用户发起了一个 HTTP 请求（HTTP 请求在网络中是明文传输的，而且必须经过运营商），于是运营商就往里塞一些内容。常见的是塞 JS 代码，我们经常看到的屏幕上方或者下方的广告就是这种情况；另外一种运营商把正常网页的内容全部塞到一个 iframe 里面，这样广告屏蔽软件就无效了，一旦屏蔽会导致全站都被屏蔽，整个网站都不能活。

那么遇到这种情况应该如何优化呢？

1. 使用 HTTPS 通信

HTTPS 在应用层增加了一个 SSL 协议，会对数据进行加密，当然加密也是有代价的，不同于 TCP/IP 的三次握手，它需要七次握手，而且加上加密、解密等因素，整个系统的性能大概会下降 10%，但这样就能基本避免运营商劫持。

2. 数据 MD5 校验

通常在生成内容时，同时会生成一个 MD5 值，就像手机 APK 防篡改一样，如果其与最终访问内容的 MD5 不匹配，就判定数据是非法的，可以认为数据被篡改了，那么将不予处理。如你的地址是 <http://xxx.xxx.xxx.xxx/room1/640.m3u8?key='xxx'>，并且其中的内容被篡改，和实际返回的请求结果不一样，即发生了劫持，这时可以对篡改内容进行过滤。

3. 数据时效性校验

通常会对数据内容进行时效性检查，如约定数据产生的时间点，客户端也有一个校验保

证。针对应用里边的页面内容，会对页面的内容以及跳转地址进行一个黑白名单的匹配，这样可以判别这些数据是否是合法的、时效性保证的。

4. 使用 HttpDNS 技术

HttpDNS 使用 HTTP 协议进行域名解析，代替现有基于 UDP 的 DNS 协议，域名解析请求直接发送到 HttpDNS 服务器，从而绕过运营商的 Local DNS，这样能够避免 Local DNS 造成的域名劫持问题和调度不精准问题。可以引入开源工程 HTTPDNSLib 或者使用阿里云的 HttpDNS 产品。

下面介绍 HttpDNS 的原理。

(1) 客户端直接访问 HttpDNS 接口，通过云解析获取业务在域名配置管理系统上配置的访问延迟最优的 IP。

(2) 客户端获取 IP 后就直接向此 IP 发送业务协议请求。以 HTTP 请求为例，通过在 Header 中指定 host 字段，向 HttpDNS 返回的 IP 发送标准的 HTTP 请求即可。

9.6.7 CDN 节点优化

回顾一下 CDN 的概念，CDN 的全称是 Content Delivery Network，即内容分发网络。其目的是通过给现有网络增加一层新的网络架构，将网站内容发布到全网部署的 CDN 节点，不同地区用户可以就近取得所访问的内容，缩短访问时间。

当用户访问已经加入 CDN 服务的网站时，首先通过 DNS 重定向技术确定最接近用户的最佳 CDN 节点，同时将用户的请求信息指向该节点。当用户的请求到达指定节点时，CDN 的服务器（节点上的高速缓存）负责将用户请求的内容提供给用户。具体流程如下：用户在自己的浏览器中输入要访问的网站的域名，浏览器向本地 DNS 请求对该域名进行解析，本地 DNS 将请求发到网站的主 DNS，主 DNS 根据一系列的策略确定当前最适合的 CDN 节点，并将解析结果（IP 地址）发给用户，用户向给定的 CDN 节点请求相应网站的内容。

可对 CDN 做如下几个方向的优化。

1. 负载均衡

负载均衡技术不仅仅应用于 CDN，其在网络的很多领域都得到了广泛的应用，如服务器的负载均衡、网络流量的负载均衡。顾名思义，网络中的负载均衡就是将网络的流量尽可能均匀地分配到几个能完成相同任务的服务器或网络节点上，由此来避免部分网络节点过载。这样既可以提高网络流量，也可以提高网络的整体性能。通常这需要一些调度策略，当有任务来到时，需要调度某个服务器以达到最优访问，主要是在链路上调优。在 CDN 中，负载均衡又分

为服务器负载均衡和服务器整体负载均衡（也被称为服务器全局负载均衡）。服务器负载均衡是指能够在性能不同的服务器之间进行任务分配，既能保证性能差的服务器不成为系统的瓶颈，也能保证性能好的服务器的资源得到充分利用。而服务器整体负载均衡发挥着重要作用，其性能高低将直接影响整个 CDN 的性能。

2. 内容复制与分发技术

内容访问响应速度取决于许多因素，如网络带宽是否稳定和流畅、传输途中的路由是否有阻塞和延时、服务器的处理速度等。一个有效的方法就是利用内容复制与分发技术，将占网站主体的大部分静态网页、图像和流媒体数据分发复制到各地的加速节点上。内容复制与分发技术极大地提高了效率。

3. 缓存技术

缓存技术主要有代理缓存服务、透明代理缓存服务、使用重定向服务的透明代理缓存服务等。通过缓存服务，用户在访问网页时可以将互联网的流量降至最低。CDN 的核心作用正是提高网络的访问速度，缓存技术同样也极大地提高了效率。

第 10 章

H.264 编码及 H.265 编码

视频编码方式就是指通过特定的压缩技术，将某个视频格式的文件转换成另一种视频格式的文件的文件的方式。H.264 和 H.265 编码是目前视频格式中用得最广泛的编码方式，H.264 创造了多参考帧、多块类型、整数变换、帧内预测等新的压缩技术，使用了更精细的分像素运动矢量（1/4、1/8）和新一代的环路滤波器，使得压缩性能大大提高，系统更加完善。H.265 是 ITU-TVCEG 继 H.264 之后所制定的新的视频编码标准。H.265 标准围绕着现有的视频编码标准 H.264，保留原来的某些技术，同时对一些相关技术加以改进。H.265 旨在在有限的带宽下传输更高质量的网络视频，仅需要原先的一半带宽即可播放相同质量的视频。

10.1 H.264 编码框架

H.264 码流文件分为两层。

（1）VCL（Video Coding Layer，视频编码层）：负责高效的视频内容表示，VCL 数据即编码处理的输出，它表示被压缩编码后的视频数据序列。

（2）NAL（Network Abstraction Layer，网络提取层）：负责以网络所要求的恰当的方式对数据进行打包和传送，是传输层。不管是在本地播放还是在网络上播放，都要通过这一层来传输。

10.2 H.264 编码原理

下面主要介绍 H.264 编码原理。

(1) H.264/AVC 并未明确表述一个编解码器如何实现，而是规定了一个编码的视频比特流的句法和该比特流的解码方法，因此在实现上有较大的灵活性。H.264 和以前的 H.261、H.263、MPEG-1、MPEG-4 等的编解码器功能模块的组成类似，不同的部分是其内部各功能模块的细节部分，H.264 编解码器的功能组成如图 10-1 所示。

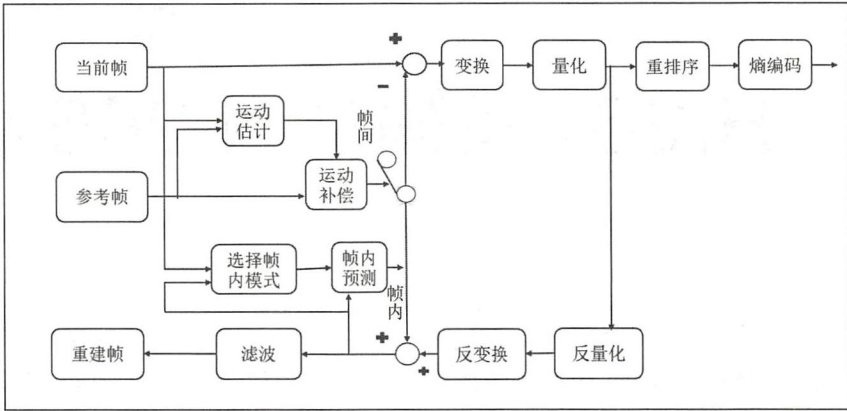


图 10-1 H.264 编解码器的功能组成

(2) H.264/AVC 编解码器的工作原理。H.264 编码器采用变换和预测混合编码方式。编码时，首先输入的帧或场 F_n 以宏块为单位被编码器处理。宏块有帧内和帧间两种模式。帧内模式使用当前帧内已编码的宏块进行预测。帧间模式使用以往一个或多个帧作为参考进行运动预测。然后，对预测值和原始值的差值进行变换、量化、重新排序和熵编码，对量化系统 X 进行逆量化、逆变换后，与预测系统相加，得到未经滤波的 uF^* 帧，对 uF^* 帧进行块间滤波，得到当前重构帧 F_n^* 。而解码过程相对比较简单，对于编码器的各部分进行逆向操作，结果经逆量化、逆变换后通过滤波器得到重构输出图像。H.264 编解码器工作原理如图 10-2 所示。

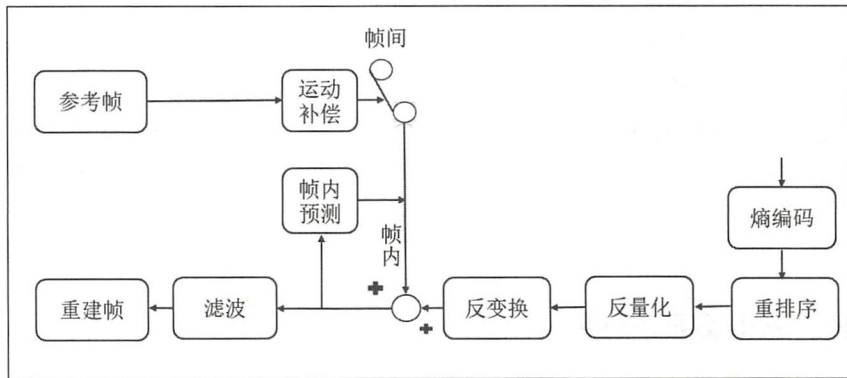


图 10-2 H.264 编解码器工作原理

10.3 H.264 码流分析

10.3.1 H.264 编码格式

在 VCL 数据传输或存储之前，这些编码的 VCL 数据先被映射或封装进 NAL 单元中。每个 NAL 单元包括一个原始字节序列负载（RBSP，Raw Byte Sequence Payload）和一组对应于视频编码的 NAL 头信息。RBSP 的基本结构：在原始编码数据的后面添加了结尾标记，一个比特“1”和若干比特“0”，以便字节对齐。H.264 码流 NAL 单元序列如图 10-3 所示。



图 10-3 H.264 码流 NAL 单元序列

10.3.2 NAL Header

NAL 头由一个字节组成，如图 10-4 所示。

语法：禁止位（1 位）、重要性指示位（2 位）、NALU 类型（5 位）。

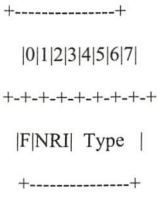


图 10-4 H.264 码流 NAL Header

NAL 头信息的每一位说明如表 10-1 所示。

表 10-1 NAL 头信息的每一位说明

位	0	1~2	3~7
简称	F	NRI	TYPE
全称	forbidden_zero_bit	nal_ref_idc	nal_unit_type
中文	禁止位	重要性指示位	NALU 类型
作用	当网络发现 NAL 单元有比特错误时，可设置该比特为 1，以便接收方丢掉该单元	标志该 NAL 单元用于重建时的重要性，值越大，越重要。取值为 00~11	1~23 表示单个 NAL 包，24~31 需要分包或者组合发送，具体含义需要参考后面的介绍

nal_unit_type 取值的含义如下：



- 0 没有定义
- (1~23 是 NAL 单元, 单个 NAL 单元包)
- 1 不分区, 非 IDR 图像的片
- 2 片分区 A
- 3 片分区 B
- 4 片分区 C
- 5 IDR 图像中的片
- 6 补充增强信息单元 (SEI)
- 7 SPS (Sequence Parameter Set, 序列参数集, 作用于一串连续的视频图像, 即视频序列)
- 8 PPS (Picture Parameter Set, 图像参数集, 作用于视频序列中的一个或多个图像)
- 9 序列结束
- 10 序列结束
- 11 码流结束
- 12 填充
- 13~23 保留
- 24 STAP-A 单一时间的组合包
- 25 STAP-B 单一时间的组合包
- 26 MTAP16 单一时间的组合包
- 27 MTAP24 多个时间的组合包
- 28 FU-A 分片的单元
- 29 FU-B 分片的单元
- 30 和 31 未定义

图 10-5 就是 H.264 码流结构。

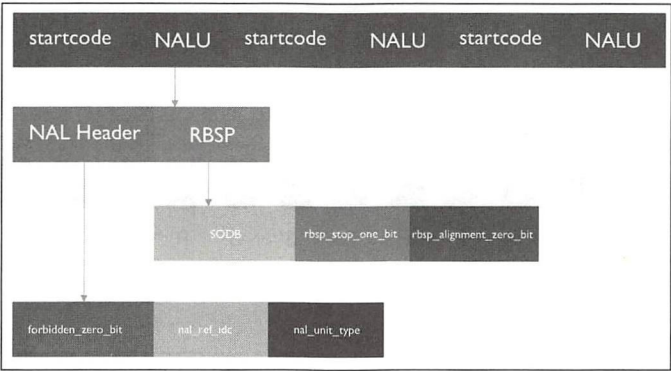


图 10-5 H.264 码流结构

10.3.3 H.264 的传输

H.264 的编码视频序列包括一系列的 NAL 单元，每个 NAL 单元包含一个 RBSP。典型的 RBSP 单元序列如图 10-6 所示。每个单元都按独立的 NAL 单元传送。NAL 单元的信息头（1 字节）定义了 RBSP 单元的类型，NAL 单元的其余部分为 RBSP 数据。

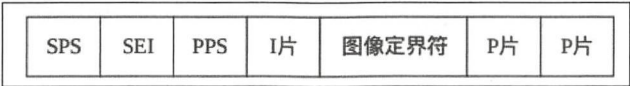


图 10-6 RBSP 单元序列

RBSP 各部分的解释如表 10-2 所示。

表 10-2 RBSP 各部分的解释

RBSP 类型	描 述
参数集 PS	序列的全局参数，如图像大小、视频格式等
增强信息 SEI	视频序列解码的增强信息
图像定界符 PD	视频图像的边界
编码	片（Slice）的头信息和数据
数据分割	DP 片层的数据，用于错误恢复解码
序列结束符	表明下一图像为 IDR 图像
流结束符	表明该流中已没有图像
填充数据	元数据，用于填充字节

10.3.4 H.264 码流结构

H.264 的功能分为两层，视频编码层（VCL）和网络提取层（NAL），VCL 数据就是被压



缩编码后的视频数据序列。在 VCL 数据封装到 NAL 单元中之后，才可以用于传输或存储。
H.264 码流 NAL 单元（也就是 NALU，Network Abstraction Layer Unit）序列如图 10-7 所示。

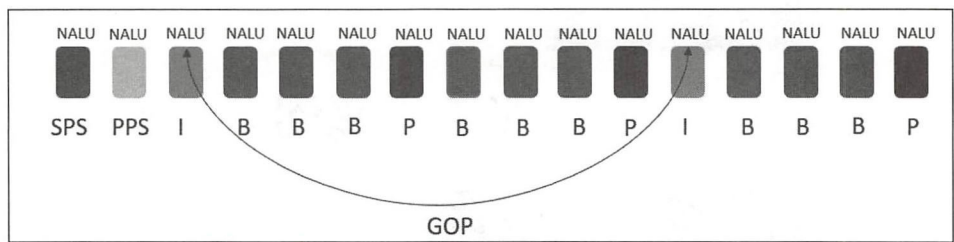


图 10-7 H.264 码流 NAL 单元序列

NALU 类型如下。

- SPS：序列参数集，作用于一系列连续的编码图像。
- PSS：图像参数集，作用于编码视频序列中一个或多个独立的图像。

参数集是一个独立的数据单位，不依赖于参数集外的其他句法元素。一个参数集不对应某一个特定的图像或序列，同一序列参数集可以被一个或者多个图像参数集引用，同理，同一个图像参数集也可以被一个或者多个图像引用。只在编码器认为需要更新参数集的内容时，才会发出新的参数集。

在复杂通信的码流中可能出现的数据单位如图 10-8 所示。

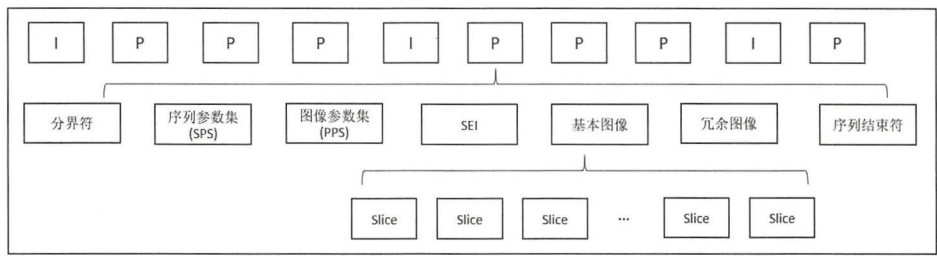


图 10-8 H.264 码流中的数据单位

用码流分析工具 Vega H264 Analyzer，打开一个 H.264 编码的 TS 流，如图 10-9 所示。

前面说到过，在 H.264 中图像以序列为单位进行组织，从码流分析工具中可以看到如下信息。

- (1) 该码流第一个序列是一个 IDR 帧。
- (2) 第一个 NALU 是 SPS，第二个 NALU 是 PPS，第三个 NALU 是 IDR。
- (3) 没有错误帧，经常在实际分析播放码流时，有些码流在编码时中间部分帧有一些错误。

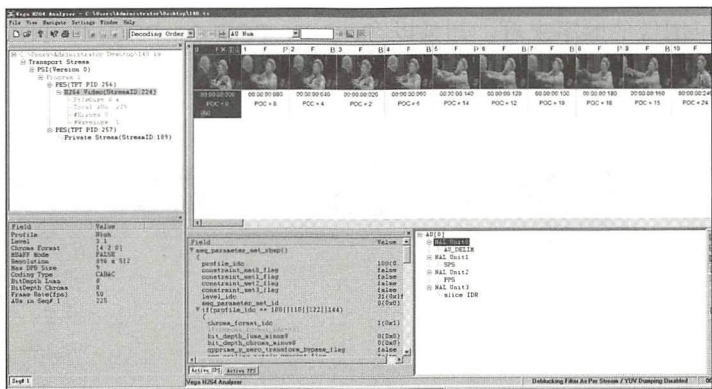


图 10-9 H.264 码流分析工具分析 TS 流

从图 10-10 中可以看到 IDR 帧，一定是 I 帧。

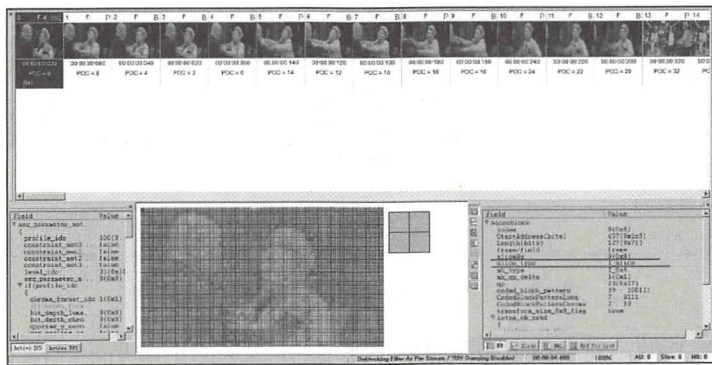


图 10-10 H.264 码流中的 IDR 帧

按解码排序，是以 POC=0，POC=8，POC=4，POC=2，POC=6 的不规则顺序排列的，可以知道解码完 I 帧后，解码 P 帧，然后是 B 帧，如图 10-11 所示。



图 10-11 H.264 码流中的解码顺序



但是按照显示图像的的顺序来看，却是以 POC=0, POC=2, POC=4, POC=6, POC=8……的顺序递增排列的，如图 10-12 所示。

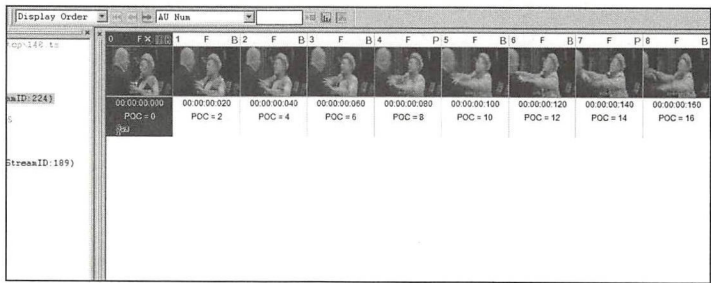


图 10-12 H.264 码流中显示图像的的顺序

这时再打开一个没有 B 帧的码流进行分析，这是一个 H.264 编码的 MP4 格式文件，如图 10-13 所示。

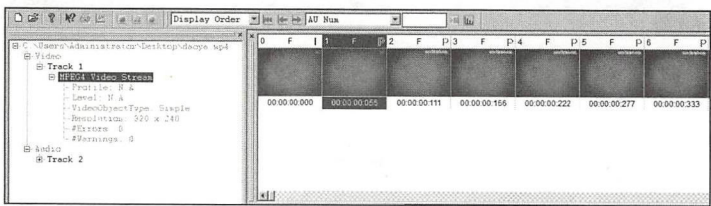


图 10-13 MP4 封装格式的 H.264 码流分析

可以看到没有 B 帧，显示图像是按照顺序递增的，如图 10-14 所示。接下来看看按照解码顺序排列的图像。

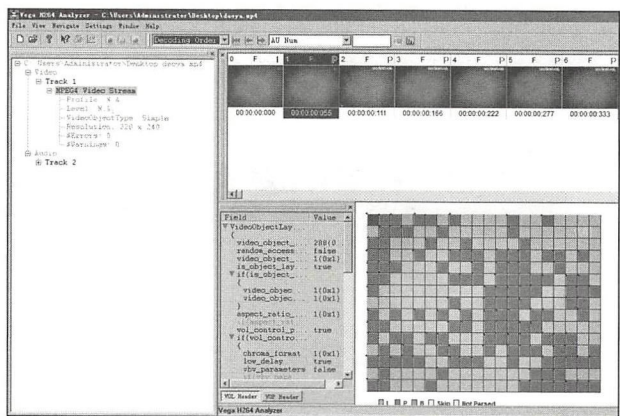


图 10-14 没有 B 帧的序列



可以发现和显示图像是一样的排列顺序，所以我们得出以下结论。

A. 视频中可以没有 B 帧。

B. 当有 B 帧时， $DTS \neq PTS$ ；当没有 B 帧时， $DTS = PTS$ 。

下面对序列进行说明。

在 H.264 协议里定义了 3 种帧，完整编码的帧叫 I 帧，参考之前的 I 帧生成的只对差异部分进行编码的帧叫 P 帧，还有一种参考前后的帧进行编码的帧叫 B 帧。在 H.264 中图像以序列为单位进行组织，一个序列是一段图像编码后的数据流，以 I 帧开始，到下一个 I 帧结束，中间部分也被称为一个 GOP。一个序列的第一个图像叫作 IDR 图像（立即刷新图像），IDR 图像都是 I 帧图像。H.264 引入 IDR 图像是为了解码的重新同步，当解码器解码到 IDR 图像时，立即将参考帧队列清空，将已解码的数据全部输出或抛弃，重新查找下一个参数集，开始解码一个新的序列。这样，如果前一个序列出现重大错误，在这里可以获得重新同步的机会。IDR 图像之后的图像永远不会使用 IDR 之前的图像的数据来解码。一个序列就是一段内容差异不太大的图像编码后生成的一串数据流。当运动变化比较少时，一个序列可以很长，因为运动变化少就代表图像画面的内容变动很小，所以就可以是一个 I 帧，然后一直是 P 帧、B 帧。当运动变化多时，一个序列可能会比较短，比如只包含一个 I 帧和几个 P 帧、B 帧。

下面对 3 种帧进行说明。

1. I 帧

I 帧指帧内编码帧，I 帧表示关键帧，你可以理解为这一帧画面的完整保留；解码时只需要本帧数据就可以完成（因为包含完整画面）。

I 帧的特点如下。

- 它是一个全帧压缩编码帧。它将全帧图像信息进行 JPEG 压缩编码及传输。
- 解码时仅用 I 帧的数据就可以重构完整图像。
- I 帧描述了图像背景和运动主体的详情。
- I 帧不需要参考其他画面生成。
- I 帧是 P 帧和 B 帧的参考帧（其质量直接影响到同组中以后各帧的质量）。
- I 帧是帧组 GOP 的基础帧（第 1 帧），在一组中只有一个 I 帧。
- I 帧不需要考虑运动矢量。
- I 帧所占数据的信息量比较大。

2. P 帧

P 帧指前向预测编码帧。P 帧表示的是这一帧跟之前的一个关键帧（或 P 帧）的差别，解



码时需要用之前缓存的画面叠加上本帧定义的差别生成最终画面。（也就是差别帧，P 帧没有完整画面数据，只有与前一帧的画面差别的数据。）

P 帧的预测与重构：P 帧以 I 帧为参考帧的，在 I 帧中找出 P 帧“某点”的预测值和运动矢量，取预测差值和运动矢量一起传送。接收端根据运动矢量从 I 帧中找出 P 帧“某点”的预测值并与差值相加以得到 P 帧“某点”的样值，从而得到完整的 P 帧。

P 帧的特点如下。

- P 帧是 I 帧后面相隔一两帧的编码帧。
- P 帧采用运动补偿的方法传送它与前面的 I 帧或 P 帧的差值及运动矢量（预测误差）。
- 解码时必须将 I 帧中的预测值与预测误差求和后才能重构完整的 P 帧图像。
- P 帧属于前向预测的帧间编码。它只参考前面最靠近它的 I 帧或 P 帧。
- P 帧可以是其后面 P 帧的参考帧，也可以是其前后 B 帧的参考帧。
- 由于 P 帧是参考帧，所以它可能造成解码错误的扩散。
- 由于是差值传送，所以 P 帧的压缩率比较高。

3. B 帧

B 帧指双向预测内插编码帧。B 帧是双向差别帧，也就是 B 帧记录的是本帧与前后帧的差别（具体比较复杂，有 4 种情况，但这样说简单些）。换言之，要解码 B 帧，不仅要取得之前的缓存画面，还要解码之后的画面，通过前后画面数据与本帧数据的叠加取得最终的画面。B 帧压缩率高，但是解码时 CPU 消耗比较大。

B 帧的预测与重构：B 帧以前面的 I 帧或 P 帧和后面的 P 帧为参考帧，“找出”B 帧“某点”的预测值和两个运动矢量，并取预测差值和运动矢量传送。接收端根据运动矢量从两个参考帧中“找出（算出）”预测值并与差值求和，得到 B 帧“某点”的样值，从而得到完整的 B 帧。

B 帧的特点如下。

- B 帧是由前面的 I 帧或 P 帧和后面的 P 帧来进行预测的。
- B 帧传送的是它与前面的 I 帧或 P 帧和后面的 P 帧之间的预测误差及运动矢量。
- B 帧是双向预测编码帧。
- B 帧压缩率最高，因为它只反映两参考帧间运动主体的变化情况，预测比较准确。
- B 帧不是参考帧，不会造成解码错误的扩散。

注意，I、B、P 各帧是根据压缩算法的需要人为定义的，它们都是实实在在的物理帧。一般来说，I 帧的压缩率是 7（跟 JPG 差不多），P 帧是 20，B 帧可以达到 50。可见使用 B 帧能



节省大量空间，节省出来的空间可以用来保存多一些 I 帧，这样在相同码率下，可以提供更好的画质。

图 10-15 为 H.264 码流分层结构，解释如下。

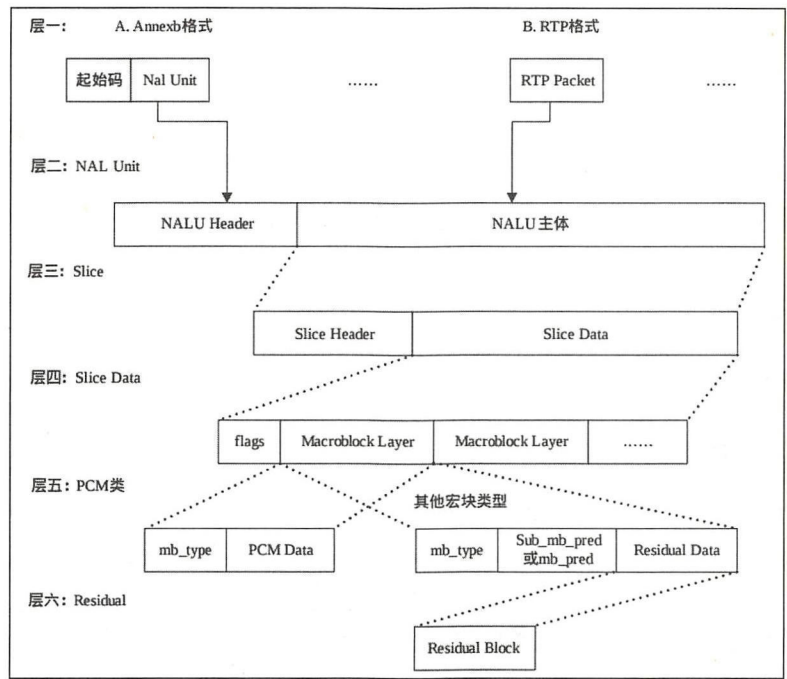


图 10-15 H.264 码流分层结构

层一：比特流，该层有两种格式，即 Annexb 格式和 RTP 格式。

层二：NAL Unit 层，包含了 NAL Header 和 NAL 主体信息。

层三：Slice 层。

1 帧视频图像可编码成一个或者多个片，每片包含整数个宏块，即每片至少一个宏块，最多时包含整个图像的宏块。

片的目的是限制误码的扩散和传输，使编码片相互间保持独立。片共有 5 种类型：I 片（只包含 I 宏块）、P 片（P 和 I 宏块）、B 片（B 和 I 宏块）、SP 片（用于不同编码流之间的切换）和 SI 片（特殊类型的编码宏块）。

片的语法结构：片头规定了片的类型、属于哪个图像、有关的参考图像等；片的数据包含了一系列宏块和不编码数据，如图 10-16 所示。



Android 音视频开发

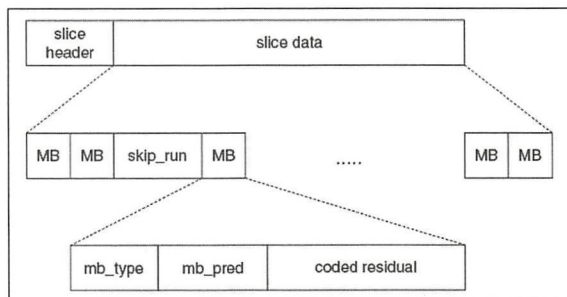


图 10-16 H.264 码流 Slice 的语法结构

层四：Slice Data 层。Slice 由宏块（Macro Block，MB）组成，宏块是编码处理的基本单元。

层五：PCM 类。

层六：Residual（残差）层。

10.3.5 H.264 的 Level 和 Profile 说明

H.264 的 Level（级别）是用来约束分辨率、帧率和码率的，具体请参照表 10-3。

表 10-3 Level 的简要说明

Level	Max macroblocks		Max video bit rate (kbit/s)				Examples for high resolution @ frame rate (max stored frames)
	per second	per frame	BP, XP, MP	HiP	Hi10P	Hi422P, Hi444PP	
1	1,485	99	64	80	192	256	128×96@30.9 (8) 176×144@15.0 (4)
1b	1,485	99	128	160	384	512	128×96@30.9 (8) 176×144@15.0 (4)
1.1	3,000	396	192	240	576	768	176×144@30.3 (9) 320×240@10.0 (3) 352×288@7.5 (2)
1.2	6,000	396	384	480	1,152	1,536	320×240@20.0 (7) 352×288@15.2 (6)
1.3	11,880	396	768	960	2,304	3,072	320×240@36.0 (7) 352×288@30.0 (6)
2	11,880	396	2,000	2,500	6,000	8,000	320×240@36.0 (7) 352×288@30.0 (6)
2.1	19,800	792	4,000	5,000	12,000	16,000	352×480@30.0 (7) 352×576@25.0 (6)



第 10 章 H.264 编码及 H.265 编码

续表

Level	Max macroblocks		Max video bit rate (kbit/s)				Examples for high resolution @ frame rate (max stored frames)
	per second	per frame	BP, XP, MP	HiP	Hi10P	Hi422P, Hi444PP	
2.2	20,250	1,620	4,000	5,000	12,000	16,000	352×480@30.7(10) 352×576@25.6 (7) 720×480@15.0 (6) 720×576@12.5 (5)
3	40,500	1,620	10,000	12,500	30,000	40,000	352×480@61.4 (12) 352×576@51.1 (10) 720×480@30.0 (6) 720×576@25.0 (5)
3.1	108,000	3,600	14,000	17,500	42,000	56,000	720×480@80.0 (13) 720×576@66.7 (11) 1280×720@30.0 (5)
3.2	216,000	5,120	20,000	25,000	60,000	80,000	1,280×720@60.0 (5) 1,280×1,024@42.2 (4)
4	245,760	8,192	20,000	25,000	60,000	80,000	1,280×720@68.3 (9) 1,920×1,080@30.1 (4) 2,048×1,024@30.0 (4)
4.1	245,760	8,192	50,000	62,500	150,000	200,000	1,280×720@68.3 (9) 1,920×1,080@30.1 (4) 2,048×1,024@30.0 (4)
4.2	522,240	8,704	50,000	62,500	150,000	200,000	1,920×1,080@64.0 (4) 2,048×1,080@60.0 (4)
5	589,824	22,080	135,000	168,750	405,000	540,000	1,920×1,080@72.3 (13) 2,048×1,024@72.0 (13) 2,048×1,080@67.8 (12) 2,560×1,920@30.7 (5) 3,680×1,536@26.7 (5)
5.1	983,040	36,864	240,000	300,000	720,000	960,000	1,920×1,080@120.5 (16) 4,096×2,048@30.0 (5) 4,096×2,304@26.7 (5)

Max macroblocks: 最大宏块数。注意，宏块大小一般是 16×16 。

- per second: 每秒（的最大宏块数），可用于约束帧率。
- per frame: 每帧（的最大帧数），可用于约束分辨率。

Android 音视频开发

Max video bit rate (kbit/s): 最大视频码率。不同档次 (Profile) 下会有区别。

- BP: Baseline Profile, 基线档次。
- XP: Extended Profile, 进阶档次。
- MP: Main Profile, 主要档次。
- HiP: High Profile, 高级档次。
- Hi10P: High 10 Profile, 高级 10 位档次。
- Hi422P: High 4:2:2 Profile, 高级 4 : 2 : 2 档次。
- Hi444PP: High 4:4:4 Predictive Profile, 高级 4 : 4 : 4 档次。

H.264 主要 Profile 说明如下。

- BP (Baseline Profile): 提供 I/P 帧, 仅支持 Progressive (逐行扫描) 和 CAVLC, 多应用于“视频会议”, 如可视电话、会议电视、远程教学、视频监控等实时通信领域。
- XP (Extended Profile): 提供 I/P/B/SP/SI 帧, 仅支持 Progressive 和 CAVLC, 多应用于流媒体领域, 如视频点播、基于网络的视频监控等。
- MP (Main Profile): 提供 I/P/B 帧, 支持 Progressive 和 Interlaced (隔行扫描), 提供 CAVLC 和 CABAC, 多应用于数字电视广播、数字视频存储等领域。
- HiP (High Profile): (Fidelity Range Extensions, FRExt) 在 Main Profile 基础上新增 8×8 帧内预测、Custom Quant、Lossless Video Coding, 以及 YUV 格式 (4:2:2, 4:4:4), 像素精度提高到 10 位或 14 位, 多应用于对分辨率和清晰度有特别要求的领域。

10.4 H.265 编码框架

10.4.1 背景知识

H.265 又被称为 HEVC (全称 High Efficiency Video Coding, 高效率视频编码, 本书统称其为 H.265), 是 ITU-T H.264/MPEG-4 AVC 标准的继任者。2004 年其由 ISO/IEC Moving Picture Experts Group (MPEG) 和 ITU-T Video Coding Experts Group (VCEG) 作为 ISO/IEC 23008-2 MPEG-H Part 2 或称作 ITU-T H.265 开始制定。第 1 版 HEVC/H.265 视频压缩标准在 2013 年 4 月 13 日被接受为国际电信联盟 (ITU-T) 的正式标准。

从 H.264 向 H.265 发展的原因如下。

(1) 视频画面要求更高，从 720 像素到 1080 像素；视频流畅度要求更高，帧率从 30fps 到 50fps

- 宏块个数的爆发式增加。
- 宏块内容复杂度的降低。
- 运动矢量数值的大幅度增加。

(2) 由于 H.264 本身的缺点

对于宏块级数字压缩视频的处理过程，H.264 一直保持 2003 年草稿发布时的实现方式，核心压缩原理没有调整与改进。

H.265 重新利用了 H.264 中定义的很多概念。两者都是基于块的视频编码技术，所以它们有着相同的根源和相近的编码方式，包括：

- 以宏块来划分图像，并最终以块来细分。
- 使用帧内压缩技术减少空间冗余。
- 使用帧内压缩技术减少时间冗余（运动估计和补偿）。
- 使用转换和量化来进行残留数据压缩。
- 使用熵编码减少残留、运动矢量传输和信号发送中的最后冗余。

事实上，视频编解码从 MPEG-1 诞生至今都没有根本性改进，H.265 也只是 H.264 在一些关键性能上的更强进化以及简单化。

10.4.2 H.265 码流结构

H.265 码流结构如图 10-17 所示。

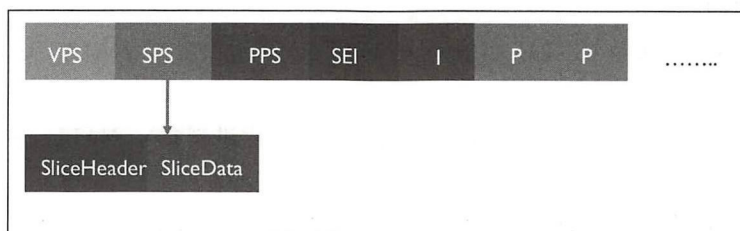


图 10-17 H.265 码流结构

1. 网络分层结构 VCL 和 NAL

与 H.264/AVC 类似，H.265/HEVC 采用了视频编码层（Video Code Layer，VCL）和网络

Android 音视频开发

适配层 (Network Abstract Layer, **NAL**), VCL 层包含了视频数据的内容, NAL 主要负责对视频压缩后的数据进行划分和封装, 保证数据能在不同的网络环境中传输。通过 NAL, 视频压缩数据将被根据其内容特性分割成具有不同特性的 NAL 单元 (NAL Unit, NALU), 并对 NALU 的内容特性进行标示。因此, 传输网络根据 NALU 的标示就可以优化视频传输的性能, 而不需要再分析视频的内容特征。NALU 可以直接作为载体进行传输, 而由于不同网络支持的最大传输单元 (Maximum Transmission Unit, MTU) 是不一样的, 因此存在一个网络分组包含一个或者多个 NALU, 或者多个网络分组包含一个 NALU。

对于一个码流文件来说, 其中包含一系列 NAL 头, 根据 H.265 对 NALU 的类型定义, 可以解析出其是 VPS、SPS、PPS 等 6 种类型。

(1) H.265 的一个图像序列的组成: VPS+SPS+PPS+SEI+一个 I 帧+若干个 P 帧。VPS、SPS、PPS、SEI、一个 I 帧、一个 P 帧都可以被称为一个 NALU。

(2) H265 的 NALU 结构: 开始码+NALU 头+NALU 数据。

- 开始码大小为 4 字节, 是一个固定值 00 00 00 01 (十六进制数), 标示一个 NALU 的开始。
- NALU 头大小为 2 字节, 共 16 位, 第 1 位值为 0, 第 2~7 位为 NALU 的 type 位 (共 6 位), 标示当前 NALU 的类型, 第 8~15 位值为 0, 第 16 位值为 1。
- NALU 数据为编码器编出来的图像信息或图像数据。

(3) 6 种类型的 NALU

- VPS (视频参数集): NALU 头值为 0x4001 (十六进制数), NALU 头 type 位值为 32 (十进制数)。
- SPS (序列参数集): NALU 头值为 0x4201 (十六进制数), NALU 头 type 位值为 33 (十进制数)。
- PPS (图像参数集): NALU 头值为 0x4401 (十六进制数), NALU 头 type 位值为 34 (十进制数)。
- SEI (补充增强信息): NALU 头值为 0x4e01 (十六进制数), NALU 头 type 位值为 39 (十进制数)。
- I 帧: NALU 头值为 0x26 01 (十六进制数), NALU 头 type 位值为 19 (十进制数)。
- P 帧: NALU 头值为 0x02 01 (十六进制数), NALU 头 type 位值为 1 (十进制数)。

(4) H.265 的 NALU 打包成 RTP 包的模式 (下面是用到的两种模式)。

- 一个 NALU 打包成一个 RTP 包, 只需要在一个 12 字节的路由头后添加去掉开始码的 NALU 即可 (这种模式在一个 NALU 的大小小于 MTU 时使用)。

- 一个 NALU 打包成几个 RTP 包 (FUs 模式), 在 12 字节的 RTP 头后面有 2 字节的 PayloadHdr 和 1 字节的 FU header。PayloadHdr 的值等于 NALU 头的 type 位改为 49 (十进制数) 后的值。FU header 第 1 位标记 RTP 包是否为 NALU 的第一片, 第 2 位标记 RTP 包是否为 NALU 的最后一片, 后 6 位是 NALU 头的 type 位。

2. 网络适配层单元 NALU

NAL 根据视频压缩数据的规则, 可以封装成不同的 NALU, NALU 包含 VPS、SPS、PPS 类型的 NALU 类型信息, 还包含视频片 (Slice) 的压缩数据。包含视频片压缩数据的 NALU, 被称为 **VCLU** (VCL NALU), 包含其他信息的压缩数据的 NALU, 则被称为 **non-VCLU** (non-VCL NALU)。H.265/HEVC 下的 NALU 包含两部分数据结构: NALU 头 (Header) 和负载 (Payload), NALU 头长度为固定的 2 字节, 反映 NALU 的内容特征, 而 NALU 负载长度为整数字节, 包含视频压缩后的原始字节序列负载 (Raw Byte Sequence Payload, RBSP)。RBSP 是对视频编码后的原始比特流片段 SODB (SString OF Data Bits) 进行添加尾部 (添加结尾比特 1, 以凑足整字节) 的包装。

RBSP 可以包含 VPS、SPS、PPS、SEI 等基本码流所需的信息, 也包含定界、序列结束、比特流结束、填充数据等。在字节流环境中, 如果 NALU 对应的 Slice 为 1 帧的开始, 则其起始码为 0x00000001; 如果对应的 Slice 不是 1 帧的开始, 则为 0x000001。为避免 NALU 负载中的字节流片段与 NALU 的起始码及结束码发生冲突, 需要对 RBSP 字节流做避免冲突处理, 经过处理后的 RBSP 才可以直接作为 NALU 的负载信息。

H.265 的 NAL 类型枚举定义如下:

```
enum NALUnitType {
    NAL_TRAIL_N    = 0,
    NAL_TRAIL_R    = 1,
    NAL_TSA_N      = 2,
    NAL_TSA_R      = 3,
    NAL_STSA_N     = 4,
    NAL_STSA_R     = 5,
    NAL_RADL_N     = 6,
    NAL_RADL_R     = 7,
    NAL_RASL_N     = 8,
    NAL_RASL_R     = 9,
    NAL_BLA_W_LP   = 16,
    NAL_BLA_W_RADL = 17,
    NAL_BLA_N_LP   = 18,
    NAL_IDR_W_RADL = 19,
    NAL_IDR_N_LP   = 20,
    NAL_CRA_NUT    = 21,
```

Android 音视频开发

```

NAL_VPS          = 32,
NAL_SPS          = 33,
NAL_PPS          = 34,
NAL_AUD          = 35,
NAL_EOS_NUT      = 36,
NAL_EOB_NUT      = 37,
NAL_FD_NUT       = 38,
NAL_SEI_PREFIX   = 39,
NAL_SEI_SUFFIX   = 40,
};

```

3. 通过工具分析 H.265 码流

对于 H.265 码流，可以通过 Elecard.HEVC.Analyzer 软件进行分析，打开一个 H.265 码流，如图 10-18 所示。

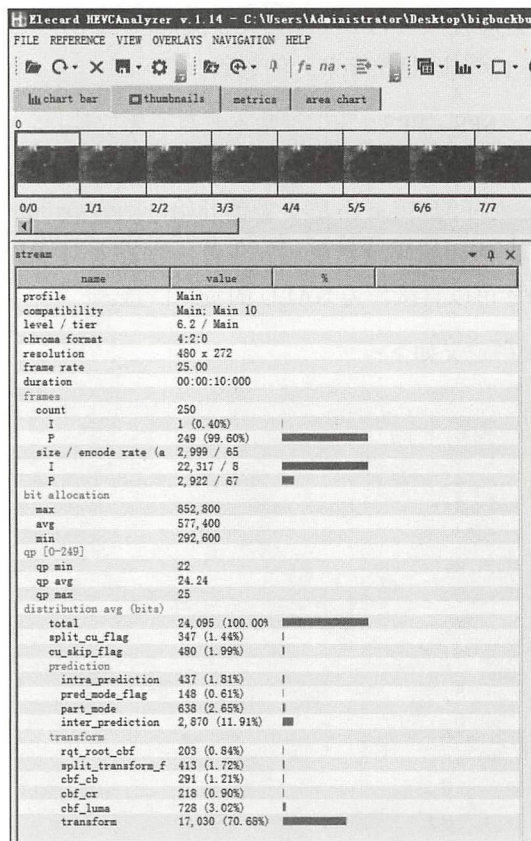


图 10-18 H.265 码流分析

从图 10-18 中可以得到如下信息。

- (1) 颜色编码为 YUV420。
- (2) 帧率为 25fps。
- (3) 250 帧，一个 I 帧，249 个 P 帧。
- (4) 分辨率为 480×272，时长 10s。

通过工具中的 stream viewer，可以看到码流信息如图 10-19 所示。

从 stream viewer 中，可以得出码流由很多个 NALU 单元组成，这些 NALU 的顺序为 VPS、SPS、PPS、IDR……展开后，每个 NALU 的内部数据结构如图 10-20 所示。

stream viewer	header	value
0x00000001	nal_unit_type = 32 (VPS_NUT) temporal_id = 0	0
0x00000001	video_parameter_set_id	3
0x00000001	reserved_zero_bits	0
0x00000001	max_layers_minus1	0
0x00000001	max_sub_layers_minus1	0
0x00000001	temporal_id_nesting_flag	1
0x00000001	reserved_zero_bits	65535
0x00000001	profile_tier_level	1
0x00000001	max_sub_layers_minus1	0
0x00000001	max_layer_id	0
0x00000001	num_layers_minus1	0
0x00000001	temporal_id_nesting_flag	0
0x00000001	video_parameter_set_id	0
0x00000001	temporal_id_nesting_flag	1
0x00000001	max_sub_layers_minus1	0
0x00000001	temporal_id_nesting_flag	0
0x00000001	video_parameter_set_id	1
0x00000001	chroma_format_idc	1
0x00000001	pic_width_in_luma_samples	480
0x00000001	pic_height_in_luma_samples	272
0x00000001	conformance_window_flag	1
0x00000001	conf_win_left_offset	0
0x00000001	conf_win_right_offset	0
0x00000001	conf_win_top_offset	0
0x00000001	conf_win_bottom_offset	0
0x00000001	bit_depth_luma_minus8	0
0x00000001	bit_depth_chroma_minus8	0
0x00000001	log2_max_pic_order_in_luma_minus4	1
0x00000001	log2_max_pic_order_in_chroma_minus4	1
0x00000001	log2_min_luma_coding_block_size_minus3	3
0x00000001	log2_min_chroma_coding_block_size_minus3	3
0x00000001	log2_diff_max_min_luma_coding_block_size	0
0x00000001	log2_diff_max_min_chroma_coding_block_size	0
0x00000001	log2_max_transform_block_size	3
0x00000001	max_transform_hierarchy_inter	2
0x00000001	max_transform_hierarchy_intra	2
0x00000001	scaling_list_enabled_flag	0
0x00000001	amp_enabled_flag	1
0x00000001	entropy_coding_mode_flag	1

stream viewer	header	value
0x00000001	nal_unit_type = 33 (SPS_NUT) temporal_id = 0	0
0x00000001	sequence_parameter_set_id	1
0x00000001	profile_idc	1
0x00000001	profile_tier_level	1
0x00000001	max_sub_layers_minus1	0
0x00000001	max_layer_id	0
0x00000001	num_layers_minus1	0
0x00000001	temporal_id_nesting_flag	0
0x00000001	video_parameter_set_id	0
0x00000001	temporal_id_nesting_flag	1
0x00000001	max_sub_layers_minus1	0
0x00000001	temporal_id_nesting_flag	0
0x00000001	video_parameter_set_id	1
0x00000001	chroma_format_idc	1
0x00000001	pic_width_in_luma_samples	480
0x00000001	pic_height_in_luma_samples	272
0x00000001	conformance_window_flag	1
0x00000001	conf_win_left_offset	0
0x00000001	conf_win_right_offset	0
0x00000001	conf_win_top_offset	0
0x00000001	conf_win_bottom_offset	0
0x00000001	bit_depth_luma_minus8	0
0x00000001	bit_depth_chroma_minus8	0
0x00000001	log2_max_pic_order_in_luma_minus4	1
0x00000001	log2_max_pic_order_in_chroma_minus4	1
0x00000001	log2_min_luma_coding_block_size_minus3	3
0x00000001	log2_min_chroma_coding_block_size_minus3	3
0x00000001	log2_diff_max_min_luma_coding_block_size	0
0x00000001	log2_diff_max_min_chroma_coding_block_size	0
0x00000001	log2_max_transform_block_size	3
0x00000001	max_transform_hierarchy_inter	2
0x00000001	max_transform_hierarchy_intra	2
0x00000001	scaling_list_enabled_flag	0
0x00000001	amp_enabled_flag	1
0x00000001	entropy_coding_mode_flag	1

图 10-19 通过 stream viewer 查看码流信息

图 10-20 H.265 码流不同 NALU 的内部数据结构

第 11 章

视频格式分析

视频格式，是音视频开发中分析问题时比较重要的技术点。比如，有一段视频播着播着没声了，不确定是解码有问题，还是编码格式有问题。那么若了解视频格式的构成，就可以用格式分析工具来查看是否有异常，或是打印其相关日志，定位问题。

11.1 MP4 格式分析

MP4 是一种常见的多媒体容器格式，对应 MPEG-4 标准，这种容器格式非常全面开放，被认为可以在其中嵌入任何形式的数据，如各种编码的视频、音频等都可以。在 MP4 文件中，媒体的描述信息与媒体数据是分开的，并且媒体数据的组织也很自由，不一定要按照时间顺序排列。同时，MP4 也支持流媒体，MP4 目前被广泛用于封装 H.264 视频和 AAC 音频，是高清视频的代表。MP4 格式的官方文件后缀名是“.mp4”，另外还有其他的以 MP4 为基础进行的扩展版本或者缩水版本的格式。

在 MP4 文件格式中，所有的内容存放于一个被称为 movie 的容器中。一个 movie 可以由多个 track 组成。每个 track 就是一个随时间变化的媒体序列，track 里的每个时间单位是一个 sample，它可以是 1 帧视频，或者一段连续的压缩音频。sample 按照时间顺序排列。其中 1 帧音频可以分解成多个音频 sample，所以音频一般用 sample 作为单位，而不用帧。在 MP4 文件格式的定义里面，用 sample 表示一个时间帧或者数据单元。

几个连续的 sample 就构成了一个 chunk。通过 Vega H264 工具分析一个 MP4 文件，如图 11-1 所示。

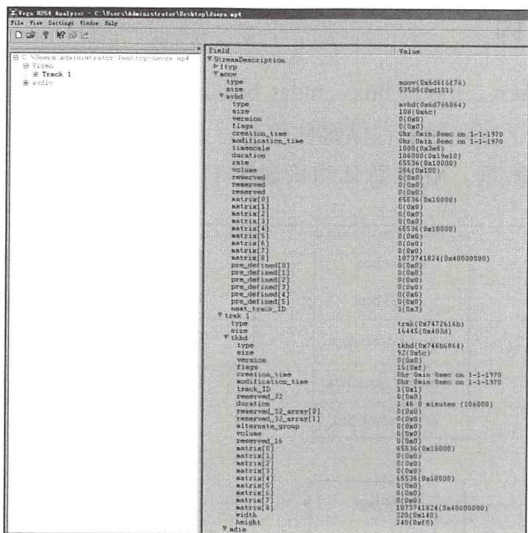


图 11-1 通过 Vega H264 工具分析一个 MP4 文件

11.1.1 Box 结构

MP4 由一个个 Box 组成，每个 Box 由一个 BoxHeader 和 BoxData 组成，BoxHeader 又可以分为如下 3 部分。

- 4 字节的 size：表示这个 Box 的大小。
- 4 字节的 type：表示这个 Box 的类型。
- 8 字节的 largesize。

如果整个 Box 的大小超出了 4 字节能表示的最大值，那么 size=1，同时 Box 的大小就存放在 largesize 中。MP4 格式 Box 结构如图 11-2 所示。

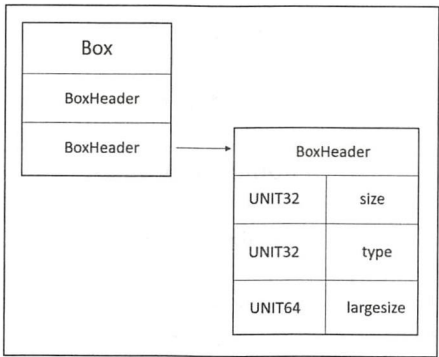


图 11-2 MP4 格式 Box 结构

11.1.2 MP4 总体结构

MP4 必须包含 ftyp box、moov box、mdat box。file type (ftyp) box 用于存储文件类型的相关信息，movie (moov) box 用于存储媒体信息，是一个 container box，media data (mdat) box 用于存储媒体的具体数据。MP4 总体结构如图 11-3 所示。

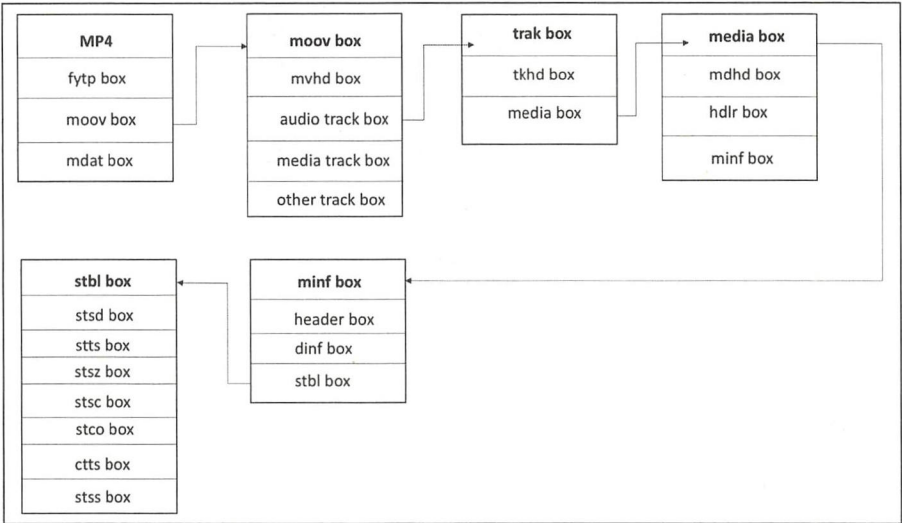


图 11-3 MP4 总体结构

11.1.3 movie (moov) box

一个 movie box 包含一个 mvhd box 和若干个 track box，比如 audio track box、media track box 等，如图 11-4 所示。



图 11-4 MP4 格式 movie box 结构

1. movie header (mvhd) box

mvhd box 具体结构如表 11-1 所示。

表 11-1 mvhd box 具体结构

字 段	字 节 数	意 义
box size	4	box 大小
box type	4	box 类型
version	1	box 版本，0 或 1，一般为 0（以下字节数均按 version=0）
flags	3	
creation time	4	创建时间（相对于 UTC 时间 1904-01-01 00:00:00 的秒数）
modification time	4	修改时间
time scale	4	文件媒体在 1s 时间内的刻度值，可以理解为 1s 长度的时间单元数
duration	4	该 track 的时间长度，用 duration 和 time scale 值可以计算 track 时长，比如 Audio track 的 time scale = 8000，duration = 560128，时长为 70.016，Video track 的 time scale = 600，duration = 42000，时长为 70
rate	4	推荐播放速率，高 16 位和低 16 位分别为小数点整数部分和小数部分，即[16.16]格式，该值为 1.0（0x00010000）时表示正常前向播放
volume	2	与 rate 类似，[8.8]格式，1.0（0x0100）表示最大音量
reserved	10	保留位
matrix	36	视频变换矩阵
pre-defined	24	
next track id	4	下一个 track 使用的 ID

2. trak box

trak box 必须包含一个 tkhd box 和一个 media box，此外还有很多可选的 box，结构如图 11-5 所示。

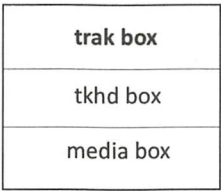


图 11-5 MP4 格式 trak box 结构

(1) track header (tkhd) box：定义了一个 track 的特性，例如时间、空间和音量信息。

tkhd box 具体结构如表 11-2 所示。

表 11-2 tkhd box 具体结构

字 段	字 节 数	意 义
box size	4	box 大小
box type	4	box 类型
version	1	box 版本，0 或 1，一般为 0（以下字节数均按 version=0）
flags	3	按位或操作结果值，预定义如下。 0x000001 track_enabled，表示该 track 是否使能播放； 0x000002 track_in_movie，表示该 track 在播放中被引用； 0x000004 track_in_preview，表示该 track 在预览时被引用。 一般该值为 7，如果一个媒体所有 track 均未设置 track_in_movie 和 track_in_preview， 将被理解为所有 track 均设置了这两项；对于 hint track，该值为 0
creation time	4	创建时间（相对于 UTC 时间 1904-01-01 00:00:00 的秒数）
modification time	4	修改时间
track id	4	ID，不能重复且不能为 0
reserved	4	保留位
duration	4	track 的时间长度
reserved	8	保留位
layer	2	视频层，默认值为 0，值小的在上层
alternate group	2	track 分组信息，默认值为 0，表示该 track 未与其他 track 有群组关系
volume	2	[8.8]格式，如果为音频，track1.0（0x0100）表示最大音量，否则为 0
reserved	2	保留位
matrix	36	视频变换矩阵
width	4	宽
height	4	高，均为[16.16]格式值

（2）media box：该 box 是一个包含一些 track 媒体数据信息 box 的 container box。

11.1.4 media box

media box 主要包含 media header（mdhd）box、hdlr box、minf box，如图 11-6 所示。



图 11-6 MP4 格式 media box 结构

1. media header (mdhd) box

media header (mdhd) box 定义了整个 movie 的特性, 例如 time scale 和 duration, mdhd box 具体结构如表 11-3 所示。

表 11-3 mdhd box 具体结构

字 段	字 节 数	意 义
box size	4	box 大小
box type	4	box 类型
version	1	box 版本, 0 或 1, 一般为 0 (以下字节数均按 version=0)
flags	3	
creation time	4	创建时间 (相对于 UTC 时间 1904-01-01 00:00:00 的秒数)
modification time	4	修改时间
time scale	4	文件媒体在 1s 时间内的刻度值, 可以理解为 1s 长度的时间单元数
duration	4	track 的时间长度
language	2	媒体语言码。最高位为 0, 后面 15 位为 3 字符 (见 ISO 639-2/T 标准中的定义)
pre-defined	2	

2. hdlr box

hdlr box 解释了媒体的播放过程信息, 同时也指明了本 track 类型, 如 video、audio、hint, 其具体结构如表 11-4 所示。

表 11-4 hdlr box 具体结构

字 段	字 节 数	意 义
box size	4	box 大小
box type	4	box 类型
version	1	box 版本, 0 或 1, 一般为 0 (以下字节数均按 version=0)
flags	3	
pre-defined	4	
handler type	4	在 media box 中, 该值为 4 字符: 'vide'——video track 'soun'——audio track 'hint'——hint track
reserved	12	
name	不定	track type name, 以\0 结尾的字符串

3. minf box

存储了解释该 track 的媒体数据的 handler-specific 信息。media handler 用这些信息将媒体时间映射到媒体数据，并进行处理。minf box 包含一个 header box，一个 data information (dinf) box 和一个 sample table (stbl) box。

(1) header box 定义颜色和图形模式信息。

(2) dinf box 解释如何定位媒体信息。

(3) stbl box 包含了 track 中 sample 的所有时间和位置信息，以及 sample 的编解码等信息。利用这个表，可以解释 sample 的时序、类型、大小以及在各自存储容器中的位置。

11.1.5 sample table (stbl) box

sample table (stbl) box 包含: sample description (std) box、time to sample (stts) box、sample size (stsz) box、sample to chunk (stsc) box、chunk offset (stco) box、composition time to sample (ctts) box、sync sample (stss) box，如图 11-7 所示。

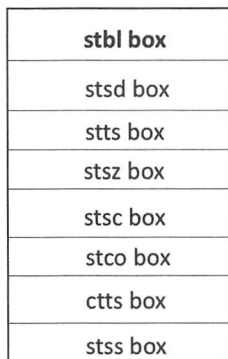


图 11-7 MP4 格式 stbl box 结构

1. sample description (std) box

box header 和 version 字段后会有一个 entry count 字段，根据 entry 的个数，每个 entry 会有 type 信息，如'vide'、'sund'等，根据不同的 type，sample description 会提供不同的信息，例如对于 video track，会有“VisualSampleEntry”类型信息，对于 audio track，会有“AudioSampleEntry”类型信息。

视频的编码类型、宽高、长度等信息，音频的声道、采样等信息，都会出现在这个 box 中。

2. time to sample (stts) box

stts 存储了 sample 的 duration, 描述了 sample 时序的映射方法, 我们通过它可以找到任何时间的 sample。stts 可以包含一个压缩的表来映射时间和 sample 序号, 用其他的表来提供每个 sample 的长度和指针。表中每个条目提供了在同一时间里面连续的 sample 序号的偏移量, 以及 sample 的偏移量。递增这些偏移量值, 就可以建立一个完整的 time to sample 表。

3. sample size (stsz) box

stsz 定义了每个 sample 的大小, 包含媒体中全部 sample 的数目和一张给出每个 sample 大小的表。这个 box 相对来说体积是比较大的。

4. sample to chunk (stsc) box

用 chunk 组织 sample 可以方便优化数据获取, 一个 chunk 包含一个或多个 sample。stsc 用一个表描述了 sample 与 chunk 的映射关系, 查看这张表就可以找到包含指定 sample 的 chunk, 从而找到这个 sample。

5. chunk offset (stco) box

stco 定义了每个 chunk 在媒体流中的位置。位置有两种可能, 32 位的和 64 位的, 后者对非常大的电影很有用。在一张表中只会有一种可能性, 这个位置是在整个文件中的, 而不是在任何 box 中的, 这样做就可以直接在文件中找到媒体数据, 而不用解释 box。需要注意的是, 一旦前面的 box 有了任何改变, 都要重新建立这张表, 因为位置信息已经改变了。

6. composition time to sample (ctts) box

ctts 提供了一个从解码时间到显示时间的 sample 一对一的映射, 具有如下的映射关系:

$$CT(n) = DT(n) + CTTS(n)$$

其中, $CTTS(n)$ 是 sample n 在 table 中的 entry (这里假设一个 entry 只对应一个 sample), 可以是正值也可以是负值; $DT(n)$ 是 sample n 的解码时间, 通过 Time-To-Sample Atoms 计算获得; $CT(n)$ 是 sample n 的显示时间。

7. sync sample (stss) box

stss 用于确定 media 中的关键帧。对于压缩媒体数据, 关键帧是一系列压缩序列的开始帧, 其解压缩时不依赖以前的帧, 而后续帧的解压缩将依赖于这个关键帧。stss 可以非常紧凑地标记媒体内的随机存取点, 它包含一个 sample 序号表, 表内的每一项严格按照 sample 的序号排列, 这说明了媒体中的哪一个 sample 是关键帧。如果此表不存在, 说明每一个 sample 都是一个关键帧, 是一个随机存取点。

11.2 FLV 格式分析

11.2.1 FLV 文件结构

FLV 文件格式主要包括 FLV Header 和 FLV Body 两部分，如图 11-8 所示。

1. Header

Header 包括文件类型、流信息（是否包含视频或音频）、Header 长度等信息。

2. Body

- Body 由一个个 Tag 组成。
- Tag 包含的信息主要有视频、音频或脚本信息。
- Body 中一般第一个 Tag 为脚本信息（MetaData，包含音视频的编码格式，视频的宽高信息等，该类型的 Tag 一般有且仅有一个），其后的 Tag 为 Video 或 Audio 的 Tag。

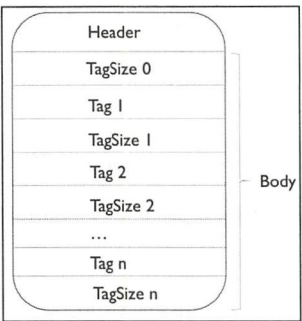


图 11-8 FLV 文件格式结构

FLV 文件格式详细扩展结构如图 11-9 所示。

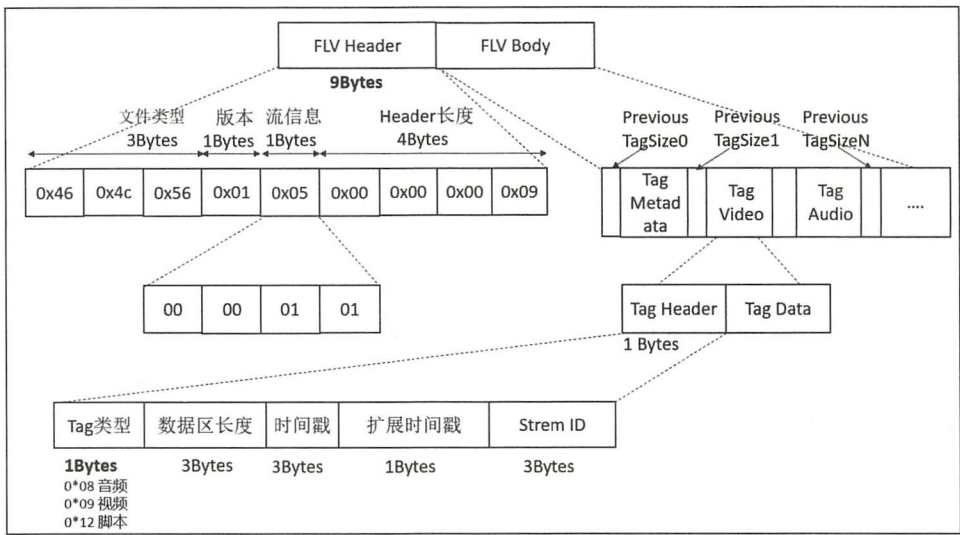


图 11-9 FLV 文件格式详细扩展结构

11.2.2 File Header（文件头）

File Header 部分记录了 FLV 的类型、版本等信息，是 FLV 的开头，一般都差不多占 9 字

节空间，其具体格式如表 11-5 所示。

表 11-5 File Header 具体格式

字 段	字 节 数	意 义
文件类型	3	固定为十六进制值 0x46、0x4c、0x56，即字符 F、L、V 的 ASCII 码
版本	1	一般为 0x01
流信息	1	倒数第 1 位是 1 表示有视频，倒数第 3 位是 1 表示有音频，倒数第 2、4 位必须为 0，比如： 00 00 01 01 表明该文件中包含视频和音频； 00 00 00 01 表明该文件中仅包含视频
Header 长度	4	整个 Header 的长度，一般为 9，大于 9 表示下面还有扩展信息

11.2.3 Body

Body 部分由一个个 Tag 组成，每个 Tag 的下面有一块 4 字节的空间，用来记录这个 Tag 的长度，这个部分放在后面用于逆向读取处理。

11.2.4 Tag

每个 Tag 也是由两部分组成的，分别是 Tag Header 和 Tag Data。Tag Header 里存放的是当前 Tag 的类型、数据区（Tag Data）长度等信息，具体如下。

1. Tag Header

Tag Header 具体格式如表 11-6 所示。

表 11-6 Tag Header 具体格式

字 段	字 节 数	意 义
TagType	1	音频为 0x08，视频为 0x09，脚本数据为 0x12
DataSize	3	数据区的长度
Timestamp	3	整数，单位为 ms，相对于第一个 Tag 的时间戳，因此第一个 Tag 的时间戳为 0。也可以将所有 Tag 的时间戳全配置为 0，解码器会自动处理
TimestampExtended	1	将时间戳扩展为 4 字节，代表高 8 位，很少用到
StreamsID	3	默认值是 0
Data	DataSize	TagType=0x08，为 AudioData；TagType=0x09，为 VideoData；TagType=0x12，为 ScriptDataObject。但该项因为内容不固定，所以在 FLV 文件中可以没有该 Tag 内容

Android 音视频开发

2. Tag Data

Tag Header 之后紧接着的就是数据区 (Tag Data)，其长度由 Tag Header 中的 DataSize 字段表明。数据区根据 Tag 类型的不同可分为 3 种，分别是音频数据、视频数据和脚本数据。

3. 脚本 Tag Data

脚本 Tag 一般至少有一个，用于存放 FLV 的 MetaData 信息，比如 duration、audiodata rate、creator、width 等。一般这是 FLV 文件的第一个 Tag，因为在知道了这些基本信息之后，才好继续进行后面的解码工作。

首先介绍一下脚本的数据类型。所有数据都是以数据类型+（数据长度）+数据的格式出现的，也就是 amf 封装，数据类型占 1 字节，数据长度要由数据类型决定，后面才是数据。

其中数据类型的种类如下。

- 0 = Number type
- 1 = Boolean type
- 2 = String type
- 3 = Object type
- 4 = MovieClip type
- 5 = Null type
- 6 = Undefined type
- 7 = Reference type
- 8 = ECMA array type
- 10 = Strict array type
- 11 = Date type
- 12 = Long string type

如果类型为 String，后面的 2 字节为字符串的长度（Long String 是 4 字节），再后面才是字符串数据；如果类型为 Number，后面的 8 字节为 Double 类型的数据；如果类型为 Boolean，后面 1 字节为 Bool 类型。

知道了这些后，再看看 FLV 中的脚本，一般开头是 0x02，表示 String 类型，后面的 2 字节为字符串长度，一般是 0x000a（'onMetaData' 的长度），再后面就是字符串 'onMetaData'，FLV 格式的文件一般都有 onMetaData 标记，在运行 ActionScript 的时候会用到。接着后面跟的是 0x08，表示 ECMA Array 类型，这个和 Map 比较相似，一个键跟着一个值，键都是 String 类型的，所以开头的 0x02 被省略了，直接跟着的是字符串的长度，然后是字符串，再就是值的类型，也就是上面介绍的那些。

4. 视频 Tag Data

视频 Tag Data 具体格式如表 11-7 所示。

表 11-7 视频 Tag Data 具体格式

字 段	字 节 数	意 义
FrameType	4	1 为关键帧, 2 为非关键帧, 3 为 H.263 的非关键帧, 4 为服务器生成关键帧, 5 为视频信息或命令帧
CodecID	4	1 为 JPEG, 2 为 H.263, 3 为 Screen Video, 4 为 On2 VP6, 5 为 On2 VP6, 6 为 Screen videoversion 2, 7 为 AVC
VideoData	n	如果 CodecID=2, 为 H263VideoPacket; 如果 CodecID=3, 为 ScreenVideopacket; 如果 CodecID=4, 为 VP6FLVVideoPacket; 如果 CodecID=5, 为 VP6FLVAlphaVideoPacket; 如果 CodecID=6, 为 ScreenV2VideoPacket; 如果 CodecID=7, 为 AVCVideoPacket

下面介绍一下 AVCVideoPacket 格式。

AVCVideoPacket 同样包括 Packet Header 和 Packet Body 两部分, 即 AVCVideoPacket Format。对于 AVC 格式的 Video, 除了第一个字节的帧类型和编码 ID, 从第二个字节开始的具体格式如表 11-8 所示。

表 11-8 AVC 格式从第二个字节开始的具体格式

字 段	字 节 数	意 义
AVCPacketType	1	AVCPacketType=0x00, 为 AVCSequence Header; AVCPacketType=0x01, 为 AVC NALU; AVCPacketType=0x02, 为 AVC end of sequence
CompostionTime	3	如果 AVCPacketType=0x01, 为相对时间戳, 其他均为 0
Data	n	负载数据, 如果 AVCPacketType=0x00, 为 AVCDecoderConfigurationRecord; 如果 AVCPacketType=0x01, 为 NALUs; 如果 AVCPacketType=0x02, 为空

AVCDecoderConfigurationRecord 详细说明如表 11-9 所示。

表 11-9 AVCDecoderConfigurationRecord 详细说明

字 段	字 节 数
cfgVersion	8
avcProfile	8
profileCompatibility	8
avcLevel	8

Android 音视频开发

续表

字 段	字 节 数
Reserved	6
lengthSizeMinusOne	2
reserved	3
numOfSPS	5
spsLength	16
sps	<i>n</i>
numOfPPS	8
ppsLength	16
pps	<i>n</i>

一般第一个视频 Tag 会封装视频编码的总体描述信息 (AVC sequence header)，也就是 AVCDeroderConfigurationRecord 结构 (于 ISO/IEC 14496-15 AVC file format 中规定)。其结构如下：

```
aligned(8) class AVCDeroderConfigurationRecord {
    unsigned int(8) configurationVersion = 1;
    unsigned int(8) AVCProfileIndication;
    unsigned int(8) profile_compatibility;
    unsigned int(8) AVCLLevelIndication;

    bit(6) reserved = '111111'b;
    unsigned int(2) lengthSizeMinusOne;

    bit(3) reserved = '111'b;
    unsigned int(5) numOfSequenceParameterSets;

    for (i=0; i< numOfSequenceParameterSets; i++) {
        unsigned int(16) sequenceParameterSetLength ;
        bit(8*sequenceParameterSetLength) sequenceParameterSetNALUnit;
    }

    unsigned int(8) numOfPictureParameterSets;

    for (i=0; i< numOfPictureParameterSets; i++) {
        unsigned int(16) pictureParameterSetLength;
        bit(8*pictureParameterSetLength) pictureParameterSetNALUnit;
    }
}
```

例如，下面加粗格式部分就是 FLV 文件中的 AVCDeroderConfigurationRecord 部分：

```

00000130h: 00 00 00 17 00 00 00 00 01 4D 40 15 FF E1 00
0A ; .....M@.J?.
00000140h: 67 4D 40 15 96 53 01 00 4A 20 01 00 05 68 E9 23 ; gM@.
膾..J ...h?
00000150h: 88 00 00 00 00 2A 08 00 00 52 00 00 00 00 00
00 ; ?...*...R.....

```

根据 AVCDecoderConfigurationRecord 结构的定义可知:

- configurationVersion = 01。
- AVCProfileIndication = 4D。
- profile_compatibility = 40。
- AVCLevelIndication = 15。
- lengthSizeMinusOne = FF: 非常重要, 是 H.264 视频中 NALU 的长度, 计算方法是 $1 + (\text{lengthSizeMinusOne} \& 3)$, NALU 的长度怎么会一直都是 4 呢? 其实这不是 NALU 的长度, 而是 NALU 中表示长度的那个字段的长度, 这个字段的长度是 4 字节 (比较绕)。
- numOfSequenceParameterSets = E1: SPS 的个数, 计算方法是 $\text{numOfSequenceParameterSets} \& 0x1F$, 结果为 1。
- sequenceParameterSetLength = 00 0A: SPS 的长度。
- sequenceParameterSetNALUnits = 67 4D 40 15 96 53 01 00 4A 20: SPS 的内容。
- numOfPictureParameterSets = 01: PPS 的个数, 结果为 1。
- pictureParameterSetLength = 00 05: PPS 的长度。
- pictureParameterSetNALUnits = 68 E9 23 88 00: PPS 的内容。

5. 音频 Tag Data

和视频 Tag Data 一样, 第一个字节是音频的信息, 格式如下。

- 音频格式 (4 位) 取值:
 - 0 = Linear PCM, platform endian
 - 1 = ADPCM
 - 2 = MP3
 - 3 = Linear PCM, little endian
 - 4 = Nellymoser 16-kHz mono
 - 5 = Nellymoser 8-kHz mono
 - 6 = Nellymoser
 - 7 = G.711 A-law logarithmic PCM

Android 音视频开发

- 8 = G.711 mu-law logarithmic PCM
- 9 = reserved
- 10 = AAC
- 11 = Speex
- 14 = MP3 8-Khz
- 15 = Device-specific sound
- 采样率（2 位）取值：
 - 0 = 5.5-kHz
 - 1 = 11-kHz
 - 2 = 22-kHz
 - 3 = 44-kHz，对于 AAC 来说此值总是 3
- 采样长度（1 位）取值：
 - 0 = snd8Bit
 - 1 = snd16Bit，压缩过的音频都是 16 位的
- 音频类型（1 位）取值：
 - 0 = sndMono
 - 1 = sndStereo，对于 AAC 来说此值总是 1

11.3 F4V 格式分析

F4V 是 Adobe 公司为了迎接高清时代而继 FLV 格式后推出的支持 H.264 的流媒体格式。它和 FLV 的主要区别在于，FLV 格式采用的是 H.263 编码，而 F4V 则支持 H.264 编码的高清晰视频，码率最高可达 50Mb/s。

由于采用 H.264 编码，与传统的 FLV 相比，F4V 在同等体积的前提下，能够实现更高的分辨率，并支持更高的比特率。随着网络带宽的发展和视频网站的发展，以及人们对视频清晰度要求越来越高，F4V 已经不断取代 FLV，流传于更大的视频网站，成为网络流媒体主流格式。

但由于 F4V 的新兴，各大视频网站采用的 F4V 标准非常多，这也决定了 F4V 相比于传统 FLV，兼容能力相对较弱。

需要注意的是，F4V 和 MP4 是兼容的格式，都属于 ISMAMP4 容器，但是 F4V 只用来封装 H.264 视频编码和 AAC 音频编码。

FLV 是 Adobe 私有格式，但是也可以用来封装 H.264 视频编码、AAC 音频编码或 H.263 视频编码、MP3 音频编码。所以不能看到后缀名为 flv 就认为不是 H.264 编码的 F4V，网络上

很多后缀名为 flv 的视频，其实质是更高清晰度的 F4V。

下面介绍 F4V 格式，F4V Box 具体格式如表 11-10 所示。

表 11-10 F4V Box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	AVCPacketType=0x00，为 AVCSequence Header；AVCPacketType=0x01，为 AVC NALU；AVCPacketType=0x02，为 AVC end of sequence
Payload	UI8[]	如果 AVCPacketType=0x01，为相对时间戳，其他均为 0
ExtendedSize	n	负载数据，如果 AVCPacketType=0x00，为 AVCDecoderConfigurationRecord；如果 AVCPacketType=0x01，为 NALUs；如果 AVCPacketType=0x02，为空

每个 Box 结构以 BoxHeader 结构开始。BoxHeader 结构字段介绍如表 11-11 所示。

表 11-11 BoxHeader 结构字段介绍

字 段	类 型	意 义
TotalSize	UI32	AVCPacketType=0x00，为 AVCSequence Header；AVCPacketType=0x01，为 AVC NALU；AVCPacketType=0x02，为 AVC end of sequence
BoxType	UI32	如果 AVCPacketType=0x01，为相对时间戳，其他均为 0
ExtendedSize	IF TotalSize == 1 UI64	负载数据，如果 AVCPacketType=0x00，为 AVCDecoderConfigurationRecord；如果 AVCPacketType=0x01，为 NALUs；如果 AVCPacketType=0x02，为空

一个 F4V Box 包含一个 ftyp box，它用于描述文件类型和通用性。

11.3.1 file type box

F4V 基于 ISO MPEG4 格式，后来又基于 Apple QuickTime 容器格式，衍生出的子集支持不同的功能，如果你的应用需要播放某个特殊文件，file type (ftyp) box 可帮助识别这些特殊文件的 Feature。ftyp box 具体格式如表 11-12 所示。

表 11-12 ftyp box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='ftyp' (0x66747970)
MajorBrand	UI32	对于 f4v MajorBrand 是'f4v'
MinorVersion	UI32	ismo 的版本号
CompatibleBrands	UI32[]	ismo、iso2、mp41 这 3 种协议中的随机一种

Android 音视频开发

11.3.2 movie box

movie (moov) box 是 F4V 文件中最有效的部分。moov box 包含一个或更多其他 box，定义了 F4V 数据结构。moov box 具体格式如表 11-13 所示。

表 11-13 moov box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='moov'
Boxes	Box[]	定义任意数目的文件结构

11.3.3 movie header box

movie header (mvhd) box 具体格式如表 11-14 所示。

表 11-14 mvhd box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='mvhd'
Version	UI8	0 或 1
Flags	UI24	0
CreationTime	Version=0 UI32 Version=1 UI64	F4V 文件创建时间
ModificationTime	Version=0 UI32 Version=1 UI64	F4V 文件修改时间
TimeScale	UI32	整个 F4V 文件的时间单元。如 100 个显示，时间单元就是 1/100s
Duration	Version=0 UI32 Version=1 UI64	整个 F4V 文件的时长，以 TimeScale 为单位
Rate	SI16.16	以十六进制数值来表示，0x0001000=1.0，正常播放速率
Volume	SI8.8	以八进制数值来表示，0x0100=1.0，最高音量
Matrix	SI32[9]	F4V 变化矩阵
NextTrackID	UI32	下一个呈现 track 的 ID

11.3.4 track box

每个 track 定义了 F4V 文件中一个单独 track 对应的信息，并且包含其他 box。track box 的格式如表 11-15 所示。

track header box

track header (tkhd) box 描述了一个 track 的主要属性。tkhd box 具体格式如表 11-16 所示。

表 11-15 track box 的格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='track'
Boxes	BOX[]	定义任意数目的 media track

表 11-16 tkhd box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='tkhd'
Version	UI8	0 或 1
Flags	UI24	0: track 可用; 1: track 是显示的一部分; 2: track 将被预览
CreationTime	Version=0 UI32 Version=1 UI64	track 创建时间
ModificationTime	Version=0 UI32 Version=1 UI64	track 上次修改时间
trackID	UI32	track 唯一识别身份
Duration	Version=0 UI32 Version=1 UI64	track 时长
Layer	SI16	trak 从前到后的排序, 对于 F4V 是 0
AlternateGroup	SI16	0
Volume	SI8.8	Audio track 是 0x0100, 非 Audio track 是 0
TransformMatrix	SI32[9]	一个固定的矩阵 {0x00010000,0,0 0,0x00010000,0 0,0,0x40000000}
Width	UI16.16	宽度
Height	UI16.16	高度

11.3.5 media box

media (mdia) box 包含 n 个 media track 的 box。mdia box 具体格式如表 11-17 所示。

表 11-17 mdia box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='tkhd'
Boxes	BOX[]	n 个定义了 media track 属性的 box

1. media header box

media header (mdhd) box 描述了一个 media track 的属性。mdhd box 具体格式如表 11-18 所示。

表 11-18 mdhd box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='mdhd'
Version	BOX[]	0 或 1
Flags	UI24	0
CreationTime	Version=0 UI32 Version=1 UI64	box 创建时间
ModificationTime	Version=0 UI32 Version=1 UI64	box 上次修改时间
TimeScale	UI32	track 时间单元
Duration	Version=0 UI32 Version=1 UI64	track 时长
Pad	UI1	0
Language	UI5[3]	语言, 用 3 个字符表示

2. handler reference box

handler reference (hdlr) box 描述了 track 中 media data 的类型。hdlr box 具体格式如表 11-19 所示。

表 11-19 hdlr box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='hdlr'
Version	BOX[]	0
Flags	UI24	0
Predefined	UI32	0
HandlerType	UI32	包含以下 4 种类型: 'video'= Video track 'soun'= Audio track 'data'= Data track 'hint'= Hint track
Name	String	tracktype 的名字, 主要用于调试时



11.3.6 media information box

media information (minf) box 包含定义了 track 媒体信息的 box。minf box 具体格式如表 11-20 所示。

表 11-20 minf box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='minf'
Boxes	BOX[]	n 个 track 相关的媒体信息的 box

1. video media header box

video media header (vmhd) box 包含 Video media 的基本信息。vmhd box 具体格式如表 11-21 所示。

表 11-21 vmhd box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='vmhd'
Version	BOX[]	0
Flags	UI24	1
GraphicMode	UI16	默认值为 0

2. sound media header box

sound media header (smhd) box 包含 Audio media 的基本信息。smhd box 具体格式如表 11-22 所示。

表 11-22 smhd box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='smhd'
Version	BOX[]	0
Flags	UI24	0
Balance	SI8.8	-1.0 = full left 0.0 = center 1.0 = full right

11.3.7 sample table box

sample table (stbl) box 包含形成一个 track 所需要的 sample 的属性。



stbl box 需要遵循以下规则排序：

```
Sample Description (stds)
Decoding Time to Sample (stss)
Sample to Chunk (stsc)
Sample Size (stsz)
Chunk Offset (stco)
```

stbl box 具体格式如表 11-23 所示。

表 11-23 stbl box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='stbl'
Boxes	BOX[]	<i>n</i> 个构成 track 的 sample

1. decoding time to sample box

decoding time to sample (stts) 定义了时间与取样的映射表。stts box 具体格式如表 11-24 所示。

表 11-24 stts box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='stts'
Version	UI8	0
Flags	UI24	0
Count	UI32	整个 STTSRECORD 的数量
Entries	STTSRECORD[count]	一个 STTSRECORD 结构数组

每个 STTSRECORD 格式如表 11-25 所示。

表 11-25 STTSRECORD 格式

字 段	类 型	意 义
SampleCount	UI32	用于 STTSRECORD 的 sample 的数目
SampleDelta	UI32	在 mdhd box 中 sample 的时长

2. composition time to sample box

composition time to sample (ctts) box 定义了 sample 和相对时间戳的映射关系表。ctts box 具体格式如表 11-26 所示。

每个 CTTSRECORD 格式如表 11-27 所示。



表 11-26 ctts box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='ctts'
Version	UI8	0
Flags	UI24	0
Count	UI32	整个 CTTSRECORD 的数量
Entries	CTTSRECORD[count]	一个 CTTSRECORD 结构数组

表 11-27 CTTSRECORD 格式

字 段	类 型	意 义
SampleCount	UI32	用于 CTTSRECORD 的 sample 的数目
SampleDelta	UI32	在 mdhd box 中 sample 的时长

3. sample to chunk box

sample to chunk (stsc) box 定义了 sample 与 chunk 之间的映射表。stsc box 具体格式如表 11-28 所示。

表 11-28 stsc box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='stsc'
Version	UI8	0
Flags	UI24	0
Count	UI32	整个 STSCRECORD 的数量
Entries	STSCRECORD[Count]	一个 STSCRECORD 结构数组

每个 STSCRECORD 格式如表 11-29 所示。

表 11-29 STSCRECORD 格式

字 段	类 型	意 义
FirstChunk	UI32	第一个 chunk
SamplePerChunk	UI32	chunk 的数目
SampleDescIndex	UI32	描述 chunk 的索引

4. sample size box

sample size (stsz) box 表示具体每个 sample 的大小。stsz box 具体格式如表 11-30 所示。



表 11-30 stsz box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='stsz'
Version	UI8	0
Flags	UI24	0
ConstantSize	UI32	如果所有 sample 都是 size，值是一个 constant size，否则就是 0
SizeCount	UI32	track 中的 sample 的数量
SizeTable	ConstantSize==0 UI32[Count]	sample 大小的表，如果 ConstantSize 不是 0，就是一个空表

5. chunk offset box

每个 sample table box 至少包含一个 chunk offset (stco/co64) box，stco/co64 box 定义了 chunk 偏移值。stco/co64 box 具体格式如表 11-31 所示。

表 11-31 stco/co64 box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='stco'或'co64'
Version	UI8	0
Flags	UI24	0
OffsetCount	UI32	偏移表中的偏移数目
Offsets	BOXTYPE='stco' UI32[OffsetCount] BOXTYPE='co64' UI64[OffsetCount]	文件中一个绝对偏移表

6. sync sample box

sync sample (stss) box 用于定义与同步相关的重要结构，比如可以快进/快退，如果 track 是 Video track，同步表都是一些关键帧。sample table (stbl) box 不包含一个 stss box，所有 track 中的 sample 都被视为同步表。stss box 具体格式如表 11-32 所示。

表 11-32 stss box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='stco'或'co64'
Version	UI8	0
Flags	UI24	0
SyncCount	UI32	同步表中的同步数目
SyncTable	UI32[SyncCount]	包含 syncCount 个数目的同步表



7. independent and disposable samples box

一个 stbl 或 traf box 包含一个 independent and disposable samples (sdtb) box，sdtb box 可以有一些 Feature，如快速向前随机访问，sdtb box 可以确认一个 sample 是否有 I 帧，并提供相关的帧信息。sdtb box 具体格式如表 11-33 所示。

表 11-33 sdtb box 具体格式

字 段	类 型	意 义
Header	BOXHEADER	BoxType='sdtb'
Version	UI8	0
Flags	UI24	0
SampleDependency	SAMPLEDEPENDENCY[]	box 末尾每个 sample 的从属信息

每个 SAMPLEDEPENDENCY 格式如表 11-34 所示。

表 11-34 SAMPLEDEPENDENCY 格式

字 段	类 型	意 义
SampleDependsOn	UI8	0: 未知的 sample dependency; 1: sample 依赖其他的 sample (不是 I 帧); 2: sample 不依赖其他的 sample (I 帧)
SamplesDependedOn	UI2	0: 不确定其他 sample 是否依赖当前 sample; 1: 其他 sample 依赖当前 sample; 2: 其他 sample 不依赖当前 sample
SampleHasRedundancy		0: 不确定 sample 是否有多余的编码; 1: sample 中有多余的编码; 2: sample 中没有多余的编码

到此 F4V 格式就分析完了。

11.4 TS 格式分析

11.4.1 TS 格式介绍

TS 的全称为 MPEG2-TS，TS 即 Transport Stream 的缩写。它是分包发送的，每一个包长 188 字节（还有 192 和 204 字节的包）。包的结构为，包头 4 字节（第 1 个字节为 0x47），负载为 184 字节。在 TS 流里可以填入很多类型的数据，如视频、音频、自定义信息等。MPEG2-TS 主要应用于实时传送的节目，比如实时广播的电视节目。MPEG2-TS 格式的特点就是要求



从视频流的任一片段开始都是可以独立解码的。简单地说，将 DVD 上 VOB 文件的前面一截剪掉（或者是将其变为已损坏的数据）就会导致整个文件无法解码，而电视节目是任何时候打开电视机都能解码（播放）的。

TS 解析需要参考 ISO/IEC 13818-1 的 2.4 Transport Stream Bitstream Requirements。TS 格式详细扩展结构如图 11-10 所示。

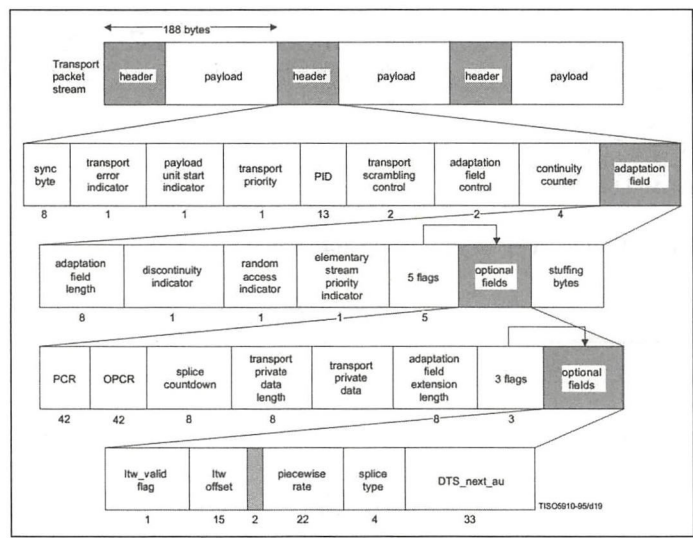


图 11-10 TS 格式详细扩展结构

11.4.2 TS 流包含的内容

一段 TS 流，必须包含 PAT 包、PMT 包、多个音频包、多个视频包、多个 PCR 包以及其他信息包。解析 TS 流数据的流程为，查找 PID 为 0x0 的包，解析 PAT，PAT 包中的 program_map_PID 表示 PMT 的 PID；查找 PMT，PMT 包中的 elementary_PID 表示音视频包的 PID，PMT 包中的 PCR_PID 表示 PCR 的 PID，有的时候 PCR 的 PID 跟音频或者视频的 PID 相同，说明 PCR 会融入音视频的包，要注意解析，而有的时候 PCR 是自己单独一个包；CAT、NIT、SDT、EIT 的 PID 分别为 0x01、0x10、0x11、0x12。

11.4.3 TS 包头解析

TS 包的包头提供关于传输方面的信息：同步、有无差错、有无加扰、PCR（节目参考时钟）等标志。TS 包的包头长度不固定，前 32 位（4 字节）固定，后面可能跟有自适应字段（适配域）。32 位（4 字节）是最小包头。包头的结构固定如图 11-11 所示。



Table 2-3 – ITU-T Rec. H.222.0 | ISO/IEC 13818 transport packet

Syntax	No. of bits	Mnemonic
transport_packet(){		
sync_byte	8	bslbf
transport_error_indicator	1	bslbf
payload_unit_start_indicator	1	bslbf
transport_priority	1	bslbf
PID	13	uimsbf
transport_scrambling_control	2	bslbf
adaptation_field_control	2	bslbf
continuity_counter	4	uimsbf
if(adaptation_field_control=='10' adaptation_field_control=='11'){		
adaptation_field()		
}		
if(adaptation_field_control=='01' adaptation_field_control=='11') {		
for (i=0;i<N;i++){		
data_byte	8	bslbf
}		
}		
}		

图 11-11 TS 格式包头结构

其中各字段的含义如下。

- sync_byte（同步字节）：固定为 0x47，该字节由解码器识别，使包头和有效负载可相互分离。
- transport_error_indicator（传输错误标志）：1 表示在相关的传输包中至少有一个不可纠正的错误位。在被置 1 后，错误被纠正之前，不能重置为 0。
- payload_unit_start_indicator（负载起始标志）：为 1 时，表示当前 TS 包的有效负载中包含 PES 或者 PSI 的起始位置；在前 4 字节之后会有一个调整字节，其数值为后面调整字段的长度 length。因此有效负载开始的位置应再偏移 1+[length]字节。
- transport_priority（传输优先级标志）：1 表明当前 TS 包的优先级比其他具有相同 PID 但此位没有被置 1 的 TS 包高。
- PID：指示存储与分组有效负载中数据的类型。PID 值 0x0000~0x000F 保留，其中 0x0000 为 PAT 保留；0x0001 为 CAT 保留；0x1fff 为分组保留，即空包。标准中定义的 PID 分配见表 11-35。

表 11-35 标准中定义的 PID 分配

PID 值	描 述
0	PAT（Program Association Table）
1	CAT（Conditional Access Table）
3~0xF	保留
0x10~0x1FFE	自定义 PID，可用于 PMT 的 PID、network 的 PID 或者其他目标
0x1FFF	空包
-	注意 PCR 的 PID 可以是 0、1 或者 0x10~0x1FFE 的任意值



- `transport_scrambling_control` (加扰控制标志): 表示 TS 流分组有效负载的加密模式。空包为 00, 如果传输包包头中包括调整字段, 不应被加密。其他取值含义是用户自定义的。
- `adaptation_field_control` (适配域控制标志): 表示包头是否有调整字段或有效负载。00 为 ISO/IEC 未来使用保留; 01 仅含有效负载, 无调整字段; 10 无有效负载, 仅含调整字段; 11 为调整字段后的有效负载, 调整字段中的前一个字节表示调整字段的长度 `length`, 有效负载开始的位置应再偏移`[length]`字节。空包应为 10。
- `continuity_counter` (连续性计数器): 随着每一个具有相同 PID 的 TS 流分组而增加, 在它达到最大值后又恢复到 0。范围为 0~15。

11.4.4 TS 包传输部分

TS 包中净荷所传输的信息包括两种类型:

- 视频、音频的 PES 包以及辅助数据。
- 节目专用信息 PSI。

当然, TS 包也可以是空包。空包用来填充 TS 流, 可以在重新进行多路复用时被插入或删除。

当系统复用时, 视频、音频的 ES 流须进行打包形成视频、音频的 PES 流, 辅助数据(如图文电视信息)不需要打成 PES 包。

11.4.5 节目专用信息 PSI (Program Specific Information)

在 TS 流中传输的主要有 4 类表格, 其中包含了解复用和显示节目相关的信息。

节目信息的结构性描述如下。

- 节目关联表 Program Association Table (PAT) 0x0000
- 节目映射表 Program Map Tables (PMT)
- 条件接收表 Conditional Access Table (CAT) 0x0001
- 网络信息表 Network Information Table (NIT) 0x0010
- 传输流描述表 Transport Stream Description Table (TSDT) 0x02

其中 PMT 中定义了与特定节目相关的 PID 信息, 比如音频包 PID、视频包 PID 以及 PCR 的 PID; CAT 表用于在流加扰情况下配置参数; NIT 是可选的, 标准中未详细定义; TSDT 也是可选的。

在将这些表信息保存到 TS 中前, 需要先切分成 section, 然后放到 TS 包中。TS 流是一种



位流（也就是数字的），它是由 ES 流分割成 PES 后复用形成的；它经过网络传输被机顶盒接收；数字电视机顶盒接收 TS 流后将解析 TS 流。

TS 流是由一个个 Packet（包）构成的，每个包都是由 Packet Header（包头）和 Packet Data（包数据）组成的。其中 Packet Header 指示了该 Packet 是什么属性的，并给出该 Packet Data 的唯一网络标识符 PID。

到这里，我们对 TS 流已经有了一定的了解，下面将学习从 TS 流转向 PAT 表和 PMT 表。

1. PAT 表（Program Association Table，节目关联表）

在 TS 流中会定期出现 PAT 表。PAT 表提供了节目号和对应的 PMT 表的 PID。其具体结构如图 11-12 所示。

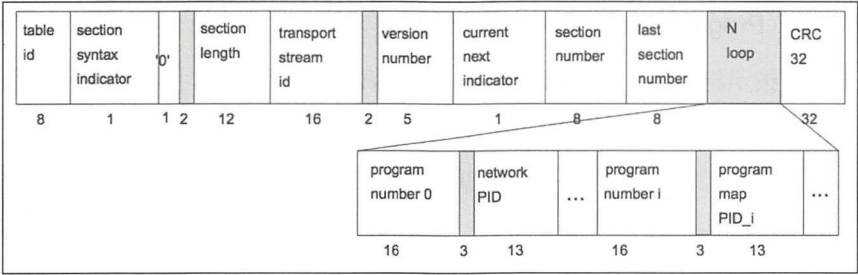


图 11-12 TS 格式 PAT 表

第 1 个字段 table id，8 位，用于标示 PSI section 负载数据的类型，其取值含义如表 11-36 所示。

表 11-36 第 1 个字段的取值含义

值	描 述
0x00	program_association_section
0x01	conditional_access_section (CA_section)
0x02	TS_program_map_section
0x03	TS_description_section
0x04	ISO_IEC_14496_scene_description_section
0x05	ISO_IEC_14496_object_descriptor_section
0x06-0x37	ITU-T Rec. H.222.0 / ISO/IEC 13818-1 reserved
0x38-0x3F	Defined in ISO/IEC 13818-6
0x40-0XFE	User private
0XFF	forbidden



PAT 表中定义的节目号（program number）与 PMT_PID 相互映射。当节目号为 0 时，存储的是 network_PID。

PAT 表定义了当前 TS 流中所有的节目，其 PID 为 0x0000，它是 PSI 的根节点，要查找节目必须从 PAT 表开始查找。PAT 表包含如表 11-37 所示的信息。

表 11-37 PAT 表包含的信息

TS 流 ID	transport_stream_id	表示 ID 标示的唯一流 ID
节目频道号	program_number	该号码标志 TS 流中的一个频道，该频道可以包含很多的节目（即可以包含多个 Video PID 和 Audio PID）
PMT 的 PID	program_map_PID	负载数据，如果 AVCPacketType=0x00，为 AVCDecoderConfigurationRecord；如果 AVCPacketType=0x01，为 NALUs；如果 AVCPacketType=0x02，为空

2. PMT 表（Program Map Table，节目映射表）

PMT 表用于指示组成某一套节目的视频、音频和数据在传送流中的位置，即对应的 TS 包的 PID 值，以及每个节目的节目时钟参考（PCR）字段的位置。

如果一个 TS 流中含有多个频道，那么就会包含多个不同 PID 的 PMT 表。

PMT 表中包含的数据如下。

- （1）当前频道中包含的所有 Video 数据的 PID。
- （2）当前频道中包含的所有 Audio 数据的 PID。
- （3）和当前频道关联在一起的其他数据的 PID（如数字广播、数据通信等使用的 PID）。

只要我们处理了 PMT 表，那么就可以获取频道中所有的 PID 信息，如当前频道包含多少个 Video，共多少个 Audio 和其他数据，还能知道每种数据对应的 PID 分别是什么。这样如果我们要选择其中一个 Video 和 Audio 收看，那么只需要把要收看的节目的 Video PID 和 Audio PID 保存下来，在处理 Packet 的时候进行过滤即可实现。其结构定义如图 11-13 所示。

其中的 stream type 标示了对应 PID 的类型，比如音频、视频或者其他类型。

3. PES 包

PES 包使用固定的 24 位起始码 0x000001 和一个 8 位的 stream-id，用于说明当前包的类型。PES 包中可以包含 DTS、PTS 等时间戳信息。整体结构如图 11-14 所示。

PES 包非定长，音频的 PES 包小于或等于 64KB，视频一般为 1 帧一个 PES 包。1 帧图像的 PES 包通常要由许多个 TS 包来传输。MPEG-2 中规定，一个 PES 包必须由整数个 TS 包来传输。如果承载一个 PES 包的最后一个 TS 包没填满，则用填充字节来填满；当下一个新的

PES 包形成时，需要用新的 TS 包来进行传输。

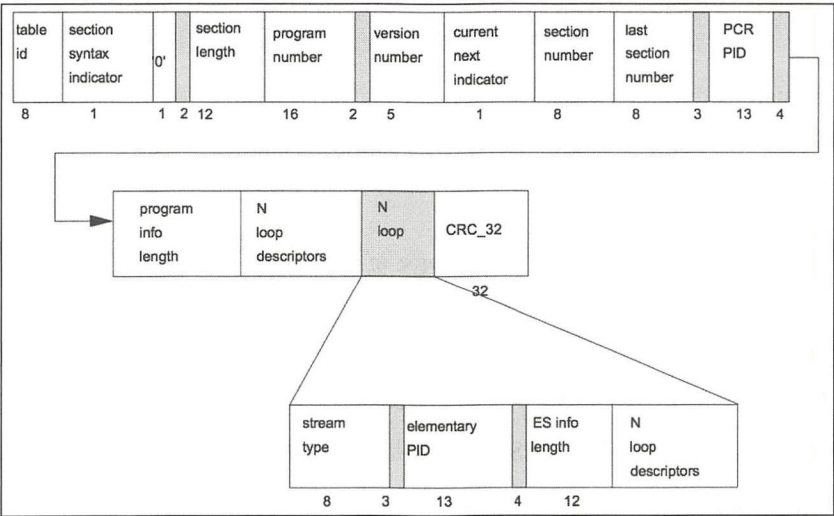


图 11-13 TS 格式 PMT 表

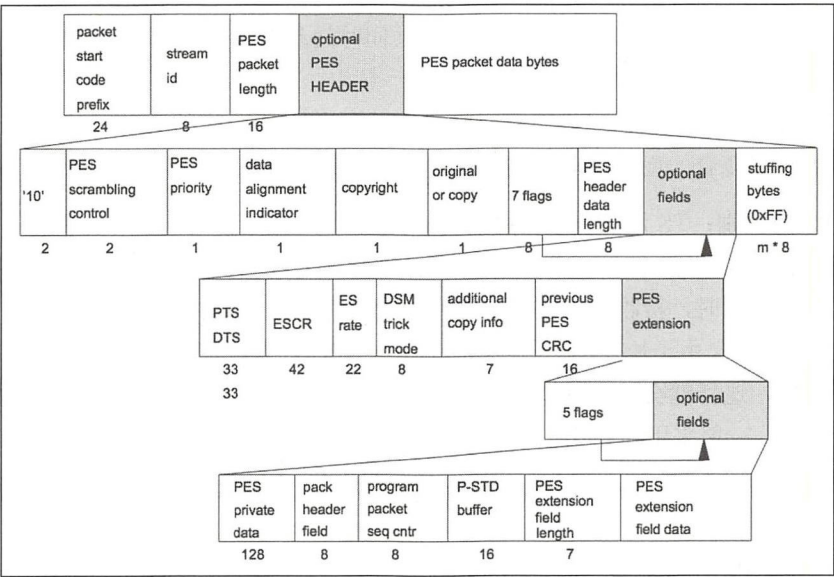


图 11-14 TS 格式 PES 包

PES 包的结构如下：

```
PES_packet() {
    packet_start_code_prefix : 24
```

```

    stream_id : 8
    PES_packet_length: 16
    optional_pes_header pes_packet_data
}

```

- packet_start_code_prefix: 24 位起始码，固定必须是'0000 0000 0000 0000 0000 0001' (0x000001)，用于标示包的开始。
- stream_id: 在 PS 流中该字段标示其存储的基本流的类型和索引号，在 TS 流中该字段仅标示其存储的基本流的类型。
- PES_packet_length: 16 位，用于存储 PES 包的长度。
- optional_pes_header 需要视 stream_id 类型而定，其长度不固定（这里包含 DTS、PTS 时间戳信息）。
- pes_packet_data 的长度是 PES_packet_length 定义的长度值。

补充一下相关知识，TS、PS、PES 和 ES 都是什么呢？

- ES (Elementary Stream): 基本码流，不分段的音频、视频或其他信息的连续码流。
- PES (Packet Elementary Stream): 把基本流 ES 分割成段，并加上相应头文件压包形成的基本码流。
- PS (Program Stream): 节目流，将具有共同时间基准的一个或多个 PES 组合（复合）而成的单一数据流。
- TS (Transport Stream): 传输流，将具有共同时间基准或独立时间基准的一个或多个 PES 组合（复合）而成的单一数据流（用于数据传输）。

TS 和 PS 的区别：TS 流的包结构是固定长度的，PS 流的包结构是可变长度的。这导致了 TS 的抵抗传输误码能力强于 PS（TS 码流由于采用了固定长度的包结构，当传输误码破坏了某一 TS 包的同步信息时，接收机可在固定的位置检测其后包中的同步信息，从而恢复同步，避免信息丢失。而 PS 包由于长度是变化的，一旦某一 PS 包的同步信息丢失，接收机无法确定下一包的同步位置，这样就会造成失步，导致严重的信息丢失。因此，当信道环境较为恶劣，传输误码率较高时，一般采用 TS 码流；而在信道环境较好，传输误码率较低时，一般采用 PS 码流。）

11.5 AVI 格式分析

AVI (Audio Video Interleaved 的缩写) 是一种 RIFF (Resource Interchange File Format 的缩写) 文件格式，多用于音视频捕捉、编辑、回放等应用程序。在通常情况下，一个 AVI 文件可以包含多个不同类型的媒体流（在典型的情况下有一个音频流和一个视频流），不过含有单一

音频流或单一视频流的 AVI 文件也是合法的。AVI 可以算是 Windows 操作系统上最基本、最常用的媒体文件格式。

先来介绍 RIFF 文件格式。RIFF 文件使用四字符码 FOURCC (four-character code) 来表征数据类型, 比如 'RIFF'、'AVI'、'LIST' 等。注意, Windows 操作系统使用的字节顺序是 little-endian, 因此一个四字符码 'abcd' 实际的 DWORD 值应为 0x64636261。另外, 四字符码中像 'AVI' 一样含有空格也是合法的。

RIFF 文件首先含有一个如图 11-15 所示的文件结构。

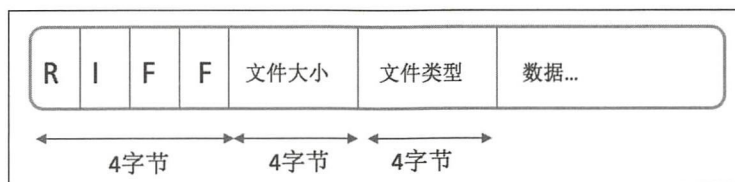


图 11-15 RIFF 文件结构

最开始的 4 字节是一个四字符码 'RIFF', 表示这是一个 RIFF 文件; 紧跟着后面用 4 字节表示此 RIFF 文件的大小; 然后又是一个四字符码说明文件的具体类型 (比如 AVI、WAVE 等); 最后就是实际的数据。注意, 文件大小值的计算方法: 实际数据长度 + 4 (文件类型域的大小), 也就是说, 文件大小的值不包括 'RIFF' 域和 “文件大小” 域本身的大小。

在 RIFF 文件的实际数据中, 通常使用列表 (List) 和块 (Chunk) 的形式来组织。列表可以嵌套子列表和块。其中, 列表的结构为 'LIST' listSize listType listData —— 'LIST' 是一个四字符码, 表示这是一个列表; listSize 占用 4 字节, 记录了整个列表的大小; listType 也是一个四字符码, 表示本列表的具体类型; listData 就是实际的列表数据。注意, listSize 值的计算方法: 实际的列表数据长度 + 4 (listType 域的大小); 也就是说 listSize 值不包括 'LIST' 域和 listSize 域本身的大小。再来看块的结构: ckID ckSize ckData —— ckID 是一个表示块类型的四字符码; ckSize 占用 4 字节, 记录了整个块的大小; ckData 为实际的块数据。注意, ckSize 值指的是实际的块数据长度, 而不包括 ckID 域和 ckSize 域本身的大小。(注意, 在下面的内容中, 将以 LIST(listType(listData)) 的形式来表示一个列表, 以 ckID(ckData) 的形式来表示一个块, 如 [optional element] 中括号中的元素表示可选项。)

11.5.1 AVI 整体结构

整个 AVI 文件是一个类型码为 'AVI' 的 RIFF 块, 其主要由 3 个 subchunk 构成: 信息块 ('hdrl' LIST 块, 用于描述 AVI 的流数据格式)、数据块 ('movi' LIST 块, 用于保存音视频序列数据)、索引块 (可选的, 'idx1' 子块)。

AVI 文件的展开结构大致如下：

```

RIFF ('AVI '
    LIST ('hdrl'
        'avih' (主 AVI 信息头数据)
        LIST ('strl'
            'strh' (流的头信息数据)
            'strf' (流的格式信息数据)
            ['strd' (可选的额外的头信息数据) ]
            ['strn' (可选的流的名字) ]
            ...
        )
        ...
    )
    LIST ('movi'
        { SubChunk | LIST ('rec '
            SubChunk1
            SubChunk2
            ...
        )
        ...
    }
    ...
)
['idx1' (可选的 AVI 索引块数据) ]
)

```

11.5.2 AVI 信息块 ('hdrl' LIST 块)

AVI 文件必需的第一个列表——'hdrl'列表，用于描述 AVI 文件中各个流的格式信息（AVI 文件中的每一路媒体数据都被称为一个流）。'hdrl'列表嵌套了一系列块和子列表——首先是一个'avih'块，其包括 AVI Main Header 和至少一个'strl' chunk（描述码流信息）。用于记录 AVI 文件的全局信息，比如流的数量、视频图像的宽和高等，可以使用一个 MainAVIHeader 结构体来操作。

MainAVIHeader 的定义如下：

```

typedef struct {
    DWORD dwMicroSecPerFrame; //视频帧的间隔时间（以 ms 为单位）
    DWORD dwMaxBytesPerSec; //这个 AVI 文件的最大数据传输速率
    DWORD dwReserved1;
    DWORD dwFlags; //AVI 文件的全局标记，比如是否含有索引块等
    DWORD dwTotalFrames; //总帧数
    DWORD dwInitialFrames; //为交互格式指定初始帧数（非交互格式应该指定为 0）

```

```

    DWORD dwStreams; //本文件包含的流的个数
    DWORD dwSuggestedBufferSize; //建议读取本文件的缓存大小（最大能容纳的块）
    DWORD dwWidth; //视频图像的宽（以像素为单位）
    DWORD dwHeight; //视频图像的高（以像素为单位）
    DWORD dwReserved[4]; //保留
} MainAVIHeader;

```

然后，就是一个或多个'strl'子列表。（文件中有多少个流，这里就对应有多少个'strl'子列表。）每个'strl'子列表至少包含一个'strh'块和一个'strf'块，而'strd'块（保存编解码器需要的一些配置信息）和'strn'块（保存流的名字）是可选的。首先是'strh'块，用于说明这个流的头信息，可以使用一个 AVIStreamHeader 结构体来操作：

```

typedef struct {
    FOURCC fccType; //必须为'strh'
    FOURCC fccHandler; //指定流的处理者，对于音视频来说就是解码器
    DWORD dwFlags; //标记：是否允许这个流输出？调色板是否有变化？
    DWORD dwPriority; //优先级
    DWORD dwInitialFrames; //指定初始帧数
    DWORD dwScale; //缩放
    DWORD dwRate;
    DWORD dwStart; //流的开始时间
    DWORD dwLength; //流的长度（单位与 dwScale 和 dwRate 的定义有关）
    DWORD dwSuggestedBufferSize; //读取这个流数据建议使用的缓存大小
    DWORD dwQuality; //流数据的质量指标（0 ~ 10000）
    DWORD dwSampleSize; //样本 size
    RECT rcFrame;
} AVIStreamHeader;

```

'strf'块，用于说明流的具体格式。如果是视频流，则使用一个 BITMAPINFO 数据结构来描述；如果是音频流，则使用一个 WAVEFORMATEX 数据结构来描述。

当 AVI 文件中的所有流都使用一个'strl'子列表说明了以后（注意，'strl'子列表出现的顺序与媒体流的编号是对应的，比如第一个'strl'子列表说明的是第一个流（Stream 0），第二个'strl'子列表说明的是第二个流（Stream 1），以此类推），'hdrl'列表的任务也就完成了，随后跟着的就是 AVI 文件必需的第二个列表——'movi'列表，用于保存真正的媒体流数据（视频图像帧数据或音频采样数据等）。

11.5.3 AVI 数据块（'movi' LIST 块）

AVI 中的信息块，包含媒体数据相关信息。那么，怎么组织这些数据呢？可以将数据块直接嵌在'movi'列表里面，也可以将数据块分组，放到一个'rec'列表中，再编排进'movi'列表。（注意，当读取 AVI 文件内容时，建议将一个'rec'列表中的所有数据块一次性读出。）



但是，当 AVI 文件中包含有多个流的时候，如何区别音频数据块与视频数据块呢？于是数据块使用了一个四字符码来表示其类型，这个四字符码由 2 字节的类型码和 2 字节的流编号组成。标准的类型码定义如下：'db'（非压缩视频帧）、'dc'（压缩视频帧）、'pc'（改用新的调色板）、'wb'（压缩音频）。比如第一个流（Stream 0）是音频，则表征音频数据块的四字符码为'00wb'；第二个流（Stream 1）是视频，则表征视频数据块的四字符码为'00db'或'00dc'。对于视频数据来说，在 AVI 数据序列中间还可以定义一个新的调色板，每个改变的调色板数据块用'xxpc'来表征，新的调色板使用数据结构 AVIPALCHANGE 来定义。（注意，如果一个流的调色板中途可能改变，则应在这个流格式的描述中，也就是在 AVISTREAMHEADER 结构的 dwFlags 中包含一个 AVISF_VIDEO_PALCHANGES 标记。）另外，文字流数据块可以使用随意的类型码表征。

11.5.4 AVI 索引块 ('idx1'子块)

紧跟在'hdr1'列表和'movi'列表之后的，就是 AVI 文件可选的索引块。这个索引块为 AVI 文件中每一个媒体数据块进行索引，并且记录它们在文件中的偏移（可能相对于'movi'列表，也可能相对于 AVI 文件头）。索引块使用一个四字符码'idx1'来表示，索引信息使用一个数据结构 AVIOLDINDEX 来定义。

```
typedef struct _avioldindex {
    FOURCC fcc; //必须为'idx1'
    DWORD cb; //本数据结构的大小，不包括最初的 8 字节（fcc 和 cb 两个域）
    struct _avioldindex_entry {
        DWORD dwChunkId; //表征本数据块的四字符码
        DWORD dwFlags; //说明本数据块是不是关键帧、是不是'rec'列表等信息
        DWORD dwOffset; //本数据块在文件中的偏移量
        DWORD dwSize; //本数据块的大小
    } aIndex[]; //这是一个数组！为每个媒体数据块都定义一个索引信息
} AVIOLDINDEX;
```

注意，如果一个 AVI 文件包含有索引块，则应在主 AVI 信息头的描述中，也就是在 AVIMAINHEADER 结构的 dwFlags 中包含一个 AVIF_HASINDEX 标记。

11.6 ASF 格式分析

11.6.1 认识 ASF

1. 什么是 ASF

ASF 是 Advance Streaming Format 的缩写，是微软为 Windows 98 所开发的串流多媒体文件



格式，也是 Windows Media 的核心。ASF 是一种数据格式，音频、视频、图像以及控制命令脚本等多媒体信息通过这种格式，以网络数据包的形式传输，实现流媒体内容发布。

2. ASF 的优势

体积小，适合网络传输。

数据组合形式灵活，可以将图形、声音和动画数据组成一个 ASF 格式文件，也可以将其他格式的视频和音频转为 ASF 格式等。

在 ASF 视频中可以带有命令代码，用户指定在到达视频或音频的某个时间后执行某个事件或操作。

3. ASF 的特点

(1) 可扩展的媒体类型——ASF 文件允许制作者定义新的媒体类型。ASF 格式灵活地定义符合 ASF 文件格式定义的新的媒体流类型。任一存储的媒体流在逻辑上都是独立于其他媒体流的，除非在文件头部分明显地定义了其与另一媒体流的关系。

(2) 插件下载——特定的有关播放插件的信息（如解压缩算法和播放器），能够存储在 ASF 文件头部分，这些信息能够为客户端找到合适的播放插件的版本（客户端没有安装这个插件）。

(3) 可伸缩的媒体类型——ASF 用于表示可伸缩媒体类型带宽之间的依赖关系的。ASF 存储各个带宽就像一个单独的媒体流。媒体流之间的依赖关系存储在文件头部分，为客户端以独立于压缩的方式解释可伸缩的选项，提供了丰富的信息流，内部的多媒体传输系统能够动态地调整以适应网络资源紧张的情况（如带宽不足）。多媒体内容的制作人要能够根据流的优先级表达他们的参考信息，如最低保证音频流的传输。随着可伸缩媒体类型的出现，流的优先级的安排变得复杂起来，因为在制作的时候很难决定各媒体流的顺序。

(4) 多语言——ASF 支持多语言。媒体流能够可选地指示所含媒体的语言。这个功能常用于音频和文本流。一个多语言 ASF 文件指的是包含不同语言版本的同一内容的一系列媒体流，其允许客户端在播放的过程中选择最合适的版本。

(5) 目录信息——目录信息既可预先定义（如作者和标题），也可以制作者自定义。目录信息功能既可以用于整个文件，也可以用于单个媒体流。

11.6.2 ASF 文件整体结构

ASF 文件是由多个 ASF Object 构成的，就像 MP4 文件是由多个 Box 构成的，ASF Object 是 ASF 的最小组成单元，其结构如表 11-38 所示。



表 11-38 ASF 文件结构

字 段	字 节	表示 ID 标示的唯一流 ID
Object GUID	16	该字段标示 TS 流中的一个频道，该频道可以包含很多节目（即可以包含多个 Video PID 和 Audio PID）
Object Size	8	负载数据，如果 AVCPacketType=0x00，为 AVCDecoderConfigurationRecord；如果 AVCPacketType=0x01，为 NALUs；如果 AVCPacketType=0x02，为空
Payload	varies	

ASF Object 的 16 字节的 GUID 代表这个 Object 的类型，Object 的类型有很多种。8 字节的 Size 代表这个 Object 的大小（=24+负载长度），紧接着就是 Object 的内容（负载）。

所有 ASF Object 和结构都是以小段字节序存储的。

1. ASF 文件顶层视图

ASF 文件通常包含 3 类 ASF Object，分别是 Header Object、Data Object 和 Index Object。

- Header Object 必须位于 ASF 文件起始位置，一个 ASF 文件中有且仅有一个 Header Object。
- Data Object 必须紧跟在 Header Object 之后，有且仅有一个。
- Index Object 是可选的，该对象主要提供了基于时间点的 ASF 随机访问机制（快速 seek）；如果 Index Object 存在，则必须是 ASF 文件的最后一个 Object。

2. Header Object

Header Object 以 ASF_Header_Object GUID Object 开头，是唯一可以包含 ASF Object 的顶层结构。它包含一系列的 ASF Object，分别代表着不同类型的 ASF Object，由 GUID 区分。它们提供了关于 ASF Data 的一些解释信息，如音视频类型及详细信息等，Header Object 通常包含如下类型的 Object。

- File Properties Object: 包含整个文件属性，如文件大小、时长等。
- Stream Properties Object: 包含 Stream 特性，如音视频格式、类型、PID 等。
- Header Extension Object: 包含用于 ASF 后向兼容的机制。
- Content Description Object: 包含内容目录信息。
- Script Command Object: 包含用于回放时执行的脚本命令。
- Marker Object: 提供文件中指定的跳转点

在一般情况下，Header Object 必须包含一个 File Properties Object、一个 Header Extension Object 和至少一个 Stream Properties Object，并且 Header Object 中 Object 出现的顺序是不固定的。



3. Index Objects

Index Objects 包含两种形式，即 Simple Index Object 和 Index Object。在 ASF 文件中 Index Object 可以有多个。

Simple Index Object 包含视频流的基于时间的索引序列，其中索引之间的间隔是固定的，通常是 1s。标准规定，对于 ASF 文件中的任意视频流必须有一个对应的 Simple Index Object，而且顺序必须以 Stream Number 出现。

Index Object 包含 Media Object Index Object 和 Timecode Index Object。Index Object 类似 Simple Index Object，保存固定时间间隔的索引序列，但可以包含非视频流；Media Object Index Object 是逐帧索引机制，支持逐帧 seek；Timecode Index Object 是基于时间码（timecode）的 seek 机制，主要针对有时间码的内容。

4. Data Object

Data Object（GUID: ASF_Data_Object）是由多个 Data Packet 构成的。这些 Data Packet 是按照发送时间的顺序排列的，并且可以来自多个不同的 Stream。Data Object 包含 64 位的 Total Data Packets 字段，用于说明当前文件包含的 Data Packet 数目。Data Packet 通常是等长的，长度固定（具体长度在 File Properties Object 中的 Data Packet Size 字段设置）。一个 Data Packet 中可以包含一个或多个 Payload。

（1）Data Packet 结构

ASF Data Packet 的结构如下：

```
=====
Error Correction Data
-----
Payload Parsing Info
-----
Payload Data
-----
Padding Data
=====
```

或者

```
=====
Error Correction Data
-----
Opaque Data
-----
Padding Data
=====
```




其中 Error Correction Data 和 Padding Data 都是可选的，不一定每一个 Data Packet 都存在这些数据。

由于 Error Correction Data 是可选的，那么如何区分 ASF Data Packet 开始位置是 Error Correction Data 还是 Payload Parsing Info 呢？主要看 Data Packet 中第 1 个字节的最高位的值（Error Correction Present），如果为 1，则是 Error Correction Data，否则是 Payload Parsing Info。不管是哪种数据，其中都存储了数据长度，可以按照 ASF Specification 文档中的 5.2 节解析。

其中，Error Correction Data 中有一个字段 Opaque Data Present，表示当前 Data Packet 结构是第二种，负载中有 Opaque Data。

（2）Payload Parsing&data 解析

Payload Parsing Info 字段的长度不是固定的，需要按照其实际字段含义解析。其中需要特别关注的是 Multiple Payloads Present、Padding Length、Send Time 字段。在解析 Payload Data 时会参考这些字段。

一个 Payload Data 可以包含一个或多个 Payload，具体需要根据 Multiple Payloads Present 的值判断，如果该值为 1，表示包含多个 Payload。

（3）Payload 结构

ASF 中定义的 Payload 如表 11-39 所示。

表 11-39 ASF 中定义的 Payload

字 段	类 型	字 节
Stream Number	BYTE	8
Media Object Number	BYTE/WORD/DWORD	0/8/16/32
Offset Into Media Object	BTYE/WORD/DWORD	0/8/16/32
Replicated Data Length	BTYE/WORD/DWORD	0/8/16/32
Replicated Data	BYTE	varies
Payload Data	BYTE	varies

其中 Stream Number 的最高位表示是否是关键帧（key frame），这也就限制了 ASF 最大能支持 128 个 Stream Media。

另外，如果 Replicated Data Length 长度为 1，则表示 Payload 是压缩格式的，具体建议参考 ASF Specification 文档中的 5.2.3 节。

Replicated Data 中包含一个 DWORD 用于标示 Payload 长度，紧跟一个 DWORD 表示时间



戳信息（以 ms 为单位），其他数据是附加的 Media sample 信息。

对于 Multiple Payloads Present 为 0 的情况，可以直接通过 Payload 的结构解析出负载数据信息。

（4）多 Payload 数据解析

在理解单个 Payload 解析之后，下面看看多 Payload 解析，其是参考 Payload Data 第 1 个字节的信息来解析的，其定义如表 11-40 所示。

表 11-40 多 Payload 解析定义

字 段	类 型	字 节
Payload Flags	BYTE	8
Number fo Payloads	-	6
Payload Length Type	-	2
Payloads	-	varies

这里的 Payload 的定义包含了 Payload Length 字段，定义结构如表 11-41 所示。

表 11-41 定义结构

字 段	类 型	字 节
Stream Number	BYTE	8
Media Object Number	BTYE/WORD/DWORD	0/8/16/32
Offset Into Media Object	BTYE/WORD/DWORD	0/8/16/32
Replicated Data Length	BTYE/WORD/DWORD	0/8/16/32
Replicated Data	BYTE	varies
Payload Length	BYTE/WORD/DWORD	0/8/16/32
Payload Data	BYTE	varies

填充字节主要为了保证 ASF Data Packet 的定长，通常填充 0 数据。

5. 如何解析 ASF MetaData

ASF 中有 5 种 Object 用于存储 MetaData，每个 MetaData 属性存储的顺序按照下面的规则。

- Content Description Object: 可用于存储长度小于 65 535 字节、与 Stream 无关、与语言无关的属性值，包括 Title、Author、Copyright、Description 和 Rating。
- Content Branding Object: 可用于存储与 Stream 无关、与语言无关的属性值 Banner Image Type、Banner Image Data、Banner Image URL 和 Copyright URL。



- Extended Content Description Object: 可用于存储任意命名的属性值, 该属性值需要长度小于 65 535, 与 Stream 无关、与语言无关, 是 WCHAR strings、BYTE arrays、Boolean values、DWORDs、QWORDs 或 WORDs 这些类型中的一种。
- MetaData Object: 可用于存储与语言无关的任意名称的属性。允许数据按照 Extended Content Description Object 格式存储。属性可以用于描述特定的 Stream, 但长度必须不超过 65 535 字节。
- MetaData Library Object: 可用于存储任意名称的属性。允许数据按照 Extended Content Description Object 格式或者 GUID 数据类型存储。属性可以用于描述特定的 Stream, 并支持指定特定语言, 并且其长度可以超过 65 535 字节 (DWORD)。

至于这 5 种 Object 如何解析, 建议参考 ASF Specification 的相关内容。

6. ASF 中的节目选择

在 ASF 文件中用 StreamID 唯一地标示一个 Stream (音频或者视频, 当然也支持其他数据, 比如字幕、脚本命令等, 目前不需要关心, 按照传统的音视频处理即可, 后续需要时可以自行按照标准处理), 这些信息可以在 Header Object 中的 Stream Properties Object 找到。比如说音频的声道数、采样率、量化位数、编码方式, 视频的编码宽高、编码方式、量化位数。

如果 ASF 文件中只有一个音频或者一个视频, 这是很容易选择的, 但是如果存在多个音频、视频文件该如何选择呢?

ASF 中提供了 Bitrate Mutual Exclusion Object、Advanced Mutual Exclusion Object、Group Mutual Exclusion Object, 用于说明不能同时播放的视频流。

Stream Prioritization Object 用于给出流播放的优先级。

Stream 选择在流媒体推流及带宽变化时用得比较多, 正常播放文件的话, 通常默认选择最大分辨率、最大码率的视频, 以及最大采样率、最多声道数的音频。

具体选择策略建议参考 ASF Specification 文档中的 7.2 节。



博文视点诚邀精锐作者加盟

十载耕耘奠定专业地位

以书为证彰显卓越品质

《C++Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博古通今、文圆质方、视角独特、点石成金之计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于IT专业出版之巔。

英雄帖

江湖风云起,代有才人出。

IT界群雄并起,逐鹿中原。

博文视点诚邀天下技术英豪加入,

指点江山,激扬文字

传播信息技术,分享IT心得

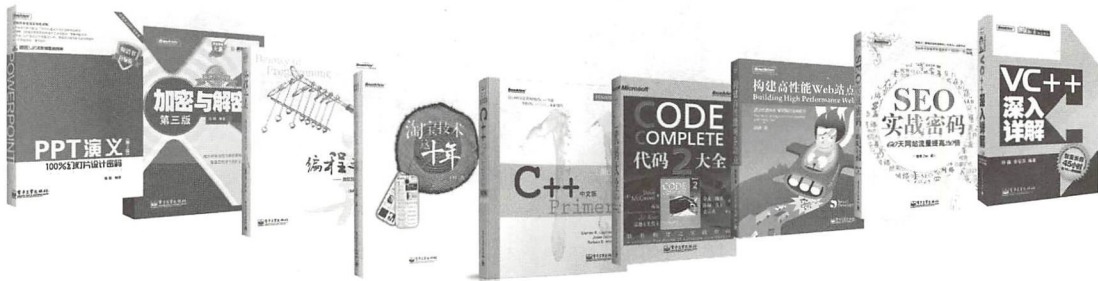
• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照IT技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身定制写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn



@博文视点Broadview



微信公众号 博文视点Broadview





博文视点精品图书展台

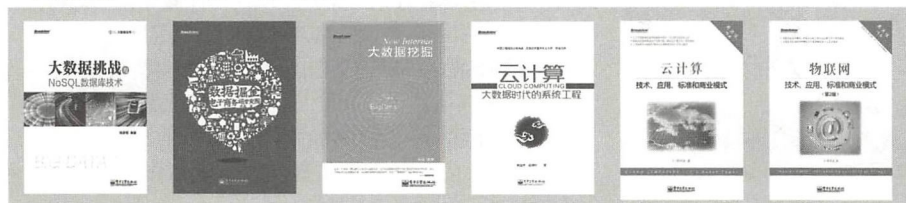
专业典藏



移动开发



大数据 · 云计算 · 物联网



数据库



Web开发



程序设计



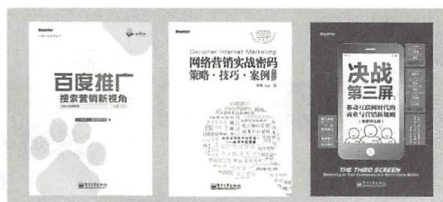
软件工程

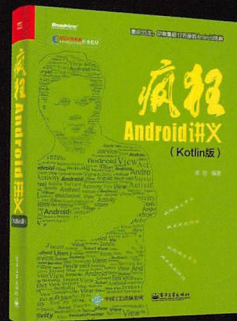
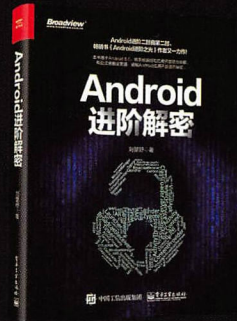
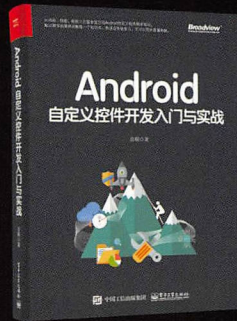


办公精品



网络营销





—— 欢迎投稿 ——

furui@phei.com.cn

@Winnie说说 (微博)

专家好评

多媒体技术是相当复杂的。你若深究，方知深不可测，必须找到一个合适的切入点。在移动互联网的时代背景下，如何快速开发音视频应用？有了本书的指引，这个问题就迎刃而解了。“Talk is cheap, show me the code.”便是作者的风格，也是本书给出的诚意满满的回答。

爱奇艺高级技术总监；陆其明

“一线经验+实例代码”，这是一本Android音视频开发技术的快速入门手册。

LiveVideoStack音视频社区创始人，包研

随着直播和短视频的兴起，音视频已经是一个非常热门的技术领域，但是这个领域却几乎没有入门书籍，本书正好填补了这一空白。书中不仅介绍了Android系统的音视频相关框架，还介绍了其他主流的开源音视频框架和流媒体技术，是入门音视频开发技术的好书。

《Android进阶之光》《Android进阶解密》作者，刘望舒

俊林兄是音视频开发领域的牛人，其公众号中分享过很多相关技术文章，而且内容比较成体系。这次他终于成功进阶为“有书人士”，这对广大读者来说是一个大好消息。本书系统地介绍了几大常用的播放器和框架，分析了它们的主体流程，而本书的后半部分还介绍了FFmpeg、直播、Codec、封装格式等内容，这些对音视频开发者来说都是很好的内容。

Powerinfo公司研发总监，许建林（Piasy）

本书内容丰富、分析透彻，从应用到原理面面俱到，是一本很好的音视频开发工具书。

腾讯音乐高级多媒体工程师，房鹏



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



责任编辑：付 睿
封面设计：罗 铭

上架建议：移动开发 > 音视频开发

ISBN 978-7-121-34996-6



9 787121 349966 >

定价：99.00元